LEARNING MADE EASY

9th Edition

Java®

for dummies®

A Wiley Brand

Create your own Java programs

Diagnose and solve coding errors

Organize data with objects and classes

Barry Burd, PhD

Author of *Beginning Programming with Java For Dummies* and *Flutter For Dummies.*

Java®

for dummies®

A Wiley Brand

Java®

9th Edition

by Barry Burd, PhD

for dummies®
A Wiley Brand

Java® For Dummies®, 9th Edition

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2025 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

Media and software compilation copyright © 2025 by John Wiley & Sons, Inc. All rights reserved, including rights for text and data mining and training of artificial technologies or similar technologies.

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. Java is a registered trademark of Oracle America, Inc. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2024951544

ISBN: 978-1-394-28924-0 (pbk); 978-1-394-28925-7 (ebk); 978-1-394-28926-4 (ebk)

Contents at a Glance

Introduction 1

Part 1: Getting Started with Java 5

CHAPTER 1: All about Java 7
CHAPTER 2: All about Software. 21
CHAPTER 3: Using the Basic Building Blocks 35

Part 2: Writing Your Own Java Programs 57

CHAPTER 4: Making the Most of Variables and Their Values 59
CHAPTER 5: Controlling the Flow by Making Decisions 97
CHAPTER 6: Controlling Program Flow with Loops 135

Part 3: Working with the Big Picture: Object-Oriented Programming 155

CHAPTER 7: The Inside scOOP 157
CHAPTER 8: Constructing New Objects. 205
CHAPTER 9: Piles of Files: Dealing with Information Overload 229
CHAPTER 10: Saving Time and Money: Reusing Existing Code 249

Part 4: Smart Java Techniques 279

CHAPTER 11: Putting Variables and Methods Where They Belong 281
CHAPTER 12: Using Collections and Streams. 313
CHAPTER 13: Using Arrays to Juggle Values 349
CHAPTER 14: Looking Good When Things Take Unexpected Turns 377
CHAPTER 15: Fancy Reference Types. 407
CHAPTER 16: Oooey GUI Was a Worm 427

Part 5: The Part of Tens 443

CHAPTER 17: Ten Packs of Java Websites 445
CHAPTER 18: Ten Bits of Advice for New Software Developers 449

Index 455

Table of Contents

INTRODUCTION 1
About This Book 1
Foolish Assumptions 2
Icons Used in This Book 3
Beyond the Book 4
Where to Go from Here 4

PART 1: GETTING STARTED WITH JAVA 5

CHAPTER 1: All about Java 7
What You Can Do with Java 8
Why You Should Use Java 9
Gaining Perspective: Where Java Fits In 10
Three Ways to Write Computer Programs 12
A word you can use to impress people 13
Imperative programming 13
Object-oriented programming 14
Functional programming 17
What's Next? 19

CHAPTER 2: All about Software 21
Get Ready for Java 21
The Inside Scoop 24
What is a compiler? 25
What is a Java virtual machine? 28
Developing Software 32
Spoiler Alert! 33

CHAPTER 3: Using the Basic Building Blocks 35
Speaking the Java Language 35
The grammar and the common names 36
The words in a Java program 37
Checking Out Java Code for the First Time 39
Understanding a Simple Java Program 41
The Java method 41
The main method in a program 43
How you finally tell the computer to do something 44
Brace yourself 46
And Now, a Few Comments 50
Adding comments to your code 51
What's Barry's excuse? 53
Using comments to experiment with your code 54

PART 2: WRITING YOUR OWN JAVA PROGRAMS 57

CHAPTER 4: Making the Most of Variables and Their Values 59

Varying a Variable 60
Assignment statements 62
The types of values that variables may have 63
How to hold the line 66
Numbers without decimal points 67
Combining declarations and initializing variables 69
Experimenting with JShell 71
The Atoms: Java's Primitive Types 74
The char type 75
The boolean type 78
Creating New Values by Applying Operators 79
Initialize once, assign often 83
The increment and decrement operators 83
Assignment operators 88
The Molecules and Compounds: Reference Types 90
Java's String type 91
Java's Random type 91
Coding with classes, objects, and their methods 92
What did he just say? 95

CHAPTER 5: Controlling the Flow by Making Decisions 97

Making Decisions (Java if Statements) 98
Guess the number 98
She controlled keystrokes from the keyboard 99
Reading double values: a cautionary tale 100
The if statement 101
Equal, equal 102
Brace yourself 103
Your intent to indent 103
Elseless in Helsinki 104
Forming Conditions with Comparisons and Logical Operators 107
Comparing numbers; comparing characters 107
Comparing objects 108
When one line isn't enough 111
Java's logical operators 112
Vive les nuls! 114
(Conditions in parentheses) 116
An import declaration 117
Warning! Explicit content! 120

The Nesting Habits of if Statements 122
Choosing among Many Alternatives 123
Java's glorious switch statement 123
Express yourself 127
Your grandparents' switch statement 129
Free fall 130

CHAPTER 6: **Controlling Program Flow with Loops** 135
Repeating Instructions Over and Over Again (Java while Statements) 136
Count On Me 139
The anatomy of a for statement 141
The world premiere of "Al's All Wet" 142
You Can Always Get What You Want 145
Who moved my file? 148
Reading a single character 148
On the var side 149
File handling in Java 150
Block on the while side 151

PART 3: WORKING WITH THE BIG PICTURE: OBJECT-ORIENTED PROGRAMMING 155

CHAPTER 7: **The Inside scOOP** 157
Defining a Class (What It Means to Be an Account) 158
Declaring variables and creating objects 160
Using an object's fields 161
One program; several classes 162
The road not taken 163
Defining a Method within a Class (Displaying an Account) 164
An account that displays itself 166
The display method's header 168
Sending Values to and from Methods (Calculating Interest) 171
Passing a value to a method 174
Returning a value from the getInterest method 175
Giving Your Numbers a Makeover 177
Hide-and-Seek 180
Good programming 181
Public lives and private dreams: Making a field inaccessible 183
Enforcing rules with accessor methods 185
Streamlining Java 186
Experiments with Variables 190
Putting a variable in its place 191
Telling a variable where to go 193

Passing Parameters. 197
Pass by value 197
Returning a result 198
Pass by reference 199
Returning an object from a method 201
A final word 203

CHAPTER 8: **Constructing New Objects** 205
Defining Constructors (What It Means to Be a Temperature) 206
What is a temperature? 207
What is a temperature scale? (Java's enum type) 207
Okay, so then what is a temperature? 208
What you can do with a temperature. 211
What does a temperature look like?. 214
Constructing a temperature; a slow-motion replay 215
The default constructor 217
The Easy Way to Create a Class. 220
Constructors don't have to be dull 223
Equality for all! 226

CHAPTER 9: **Piles of Files: Dealing with Information Overload** 229
What It Means to Be an Employee 230
Putting your class to good use 232
She reads keystrokes from the keyboard 234
Working with Disk Files. 237
Cutting a check. 241
Repeat after me 242
Reading from a file 243
Where's your file? 244
You can decide where to put your file 245
Reading a line at a time 245
Clean up after yourself 247

CHAPTER 10: **Saving Time and Money: Reusing Existing Code** 249
Defining Subclasses (What It Means to Be a Full-Time or Part-Time Employee). 250
Creating a subclass 253
Creating subclasses is habit-forming 257
Using Subclasses 258
Making types match 259
The second half of the story 260

Changing the Payments for Only Some of the Employees 262
A Java annotation. 263
Using methods from classes and subclasses 264
Yet Another switch 268
Doing Something about the Weather. 270
Constructors for subclasses. 272
Using all this stuff 273

PART 4: SMART JAVA TECHNIQUES 279

CHAPTER 11: Putting Variables and Methods Where They Belong. 281
Making a Package 282
Package deal. 282
Using the Player class 284
Running the code. 287
One class; nine objects 288
Some notes on formatting 289
It's All About Access. 291
Access for Java modules 292
Access modifiers for classes 293
A word or two about method-local variables 294
Access for members (fields and methods) 295
Making Static (Finding the Team Average). 301
So much static!. 303
Meet the compact constructor 303
Displaying the overall team average 304
All about static import 306
All about static references 307
Why static members don't always play well with others 308

CHAPTER 12: Using Collections and Streams 313
Using an ArrayList 314
A new way to do looping 318
Import a module in one fell swoop. 319
Using generics 320
Wrap it up 321
Are we done yet? 323
Once and again 324
Sorting and Searching. 324
Another Kind of Collection Class 326
So many collection classes! 328
Functional Programming 329
Problem-solving the old-fashioned way. 330
A familiar problem. 331

Lambda expressions 332
The interpretation of streams 334
Why bother? 340
Method references 343
Some black sheep among the lambdas 344

CHAPTER 13: **Using Arrays to Juggle Values** 349
Getting Your Ducks All in a Row 349
How to book hotel guests 351
Tab stops and other special things 352
Make life easy for yourself 353
Array versus ArrayList: The Ultimate Showdown 356
Do You Have a Room? 357
Writing to a file 359
What else can go wrong? 360
Array Madness 361
Return to Sanity 363
Arrays of Objects 365
Using the Room class 367
The conditional operator 370
How to Argue with Your Code 371
Settling the argument 372
Checking for the right number of program arguments 374

CHAPTER 14: **Looking Good When Things Take Unexpected Turns** 377
Garbage In 378
Making things right 382
The parameter in a catch clause 382
Do it yourself 385
Who will catch the exception? 387
Catching two or more exceptions at a time 394
The Buck Stops Here, Except When It Doesn't 395
Catch it soon 396
Catch it later 398
Checked or unchecked? 400
Try, Try Again! 402

CHAPTER 15: **Fancy Reference Types** 407
Java's Types 407
The Java Interface 408
Declaring two interfaces 409
Implementing interfaces 411
Putting the pieces together 413

Abstract Classes 417
Caring for your pet 420
Using all your classes 422
Relax! You're Not Seeing Double! 424

CHAPTER 16: **Oooey GUI Was a Worm** 427
Getting Started with GUI Applications 428
Setting up your computer 428
Creating a bare-bones JavaFX project 431
Running your bare-bones JavaFX project 434
Adding Stuff to Your JavaFX Project 436
Where's my Scene Builder application? 436
Building a scene 437
Tweaking some code 440
Running your new project 441

PART 5: THE PART OF TENS 443

CHAPTER 17: **Ten Packs of Java Websites** 445
This Book's Website 445
For Business Issues Related to This Book 445
Download the Java Development Kit 446
Your Grandparents' Java Download Site 446
The Horse's Mouth 446
Join Java User Groups 446
Find the Latest News about Java 446
Find News, Reviews, and Sample Code 447
Got a Technical Question about Anything? 447
Become Involved in the Future of Java 447

CHAPTER 18: **Ten Bits of Advice for New Software Developers** 449
How Long Does It Take to Learn Java? 450
Which of Your Books Should I Read? 450
Are Books Other than Yours Good for Learning Java? 450
Which Computer Programming Language(s) Should I Learn? 451
Which Skills Other than Computer Coding Should I Learn? 451
How Should I Continue My Learning as a Software Developer? 452
How Else Should I Continue My Learning as a Developer? 452
How Can I Land a Job Developing Software? 453
I Still Don't Know What to Do with My Life 453
If I Have Other Questions, How Can I Contact You? 454

INDEX 455

Introduction

What's all the fuss about Java? To help answer that question, I offer a few facts:

- Nine million of the world's developers work on applications in Java, and Java runs on approximately 7 billion devices.*
- Ninety percent of all Fortune 500 companies use Java.**
- According to the TIOBE Programming Community Index, Java is the world's third most popular programming language.***
- In 2021, Glassdoor, Inc., ranked jobs based on earnings potential, job satisfaction, and number of available job openings. Among the company's "50 Best Jobs in America for 2021," a career as a Java developer ranked number one.****

Sounds good. Right?

Please, read on.

About This Book

This book isn't the usual dry techie guide. It's written for normal human beings — people with little or no programming experience. In this book, I divide Java into manageable chunks. Each chunk is (more or less) a chapter on its own. I explain concepts in plain language using complete code examples that you can download and run. I keep each code example focused on a few key concepts. I resist the urge to use fancy tricks that impress professional programmers. I expand on concepts that may be difficult for newcomers. I add diagrams to help you visualize important ideas. I provide exercises with each chapter along with solutions to the exercises on the book's website.

---

* `www.geeksforgeeks.org/reasons-to-learn-java`

** `https://softjourn.com/insights/is-java-still-used`

*** `www.tiobe.com/tiobe-index`

**** `www.glassdoor.com/blog/best-jobs-in-america-for-2021`

Finally, and most importantly — and without question the most significant of all this book's features — I throw in some jokes. I've written some good jokes and lots of bad jokes. (I should say "lots and lots" of bad jokes.) I've hidden Easter eggs in the text. I've added anecdotes about all kinds of topics. Some of the anecdotes are true, and many of them are . . . well, you figure it out.

Foolish Assumptions

In this book, I make a few assumptions about you, the reader. If one of these assumptions is incorrect, you're probably okay. If all these assumptions are incorrect, please buy the book anyway:

- **I assume that you have access to a computer.** Here's the good news: You can run most of the code in this book on almost any computer. The only computers you can't use to run this code are ancient boxes that are more than ten years old (give or take a few years).
- **I assume that you can navigate your computer's common menus and dialog boxes.** You don't have to be a Windows, Linux, or Macintosh power user, but you should be able to start a program, find a file, put a file into a certain directory — that sort of thing. Most of the time, when you follow instructions in this book, you're typing code on the keyboard, not pointing-and-clicking the mouse.
- **I assume that you can think logically.** That's all there is to programming in Java — thinking logically. If you can think logically, you have it made. If you don't believe that you can think logically, read on. You may be pleasantly surprised.
- **I make few assumptions about your computer programming experience (or your lack of such experience).** In writing this book, I've tried to do the impossible: Make the book interesting for experienced programmers yet accessible to people with little or no programming experience. So, I assume no particular programming background on your part. If you've never created a loop or indexed an array, that's okay.

  On the other hand, if you've done these things (maybe in Visual Basic, Python, or C++), you'll discover some interesting plot twists in Java. The developers of Java took the best ideas in object-oriented programming, streamlined them, reworked them, and reorganized them into a sleek, powerful way of thinking about problems. You'll find many new, thought-provoking features in Java. As you find out about these features, many of them will seem quite natural to you. One way or another, you'll feel good about using Java.

Icons Used in This Book

If you could watch me write this book, you'd see me sitting at my computer, talking to myself. I say each sentence in my head. Most of the sentences, I mutter several times. When I have an extra thought or a side comment that doesn't belong in the regular stream, I twist my head a little bit. That way, whoever's listening to me (usually, nobody) knows that I'm off on a momentary tangent.

Of course, in print, you can't see me twisting my head. I need some other way to set a side thought in a corner by itself. I do it with icons. When you see a Tip icon or a Remember icon, you know that I'm taking a quick detour.

Here's a list of icons that I use in this book:

TIP

A tip is an extra piece of information — a helpful tidbit that the other books may forget to tell you.

WARNING

Everyone makes mistakes. Heaven knows that I've made a few in my time. Anyway, when I think people are especially prone to make a mistake, I mark it with a Warning icon.

REMEMBER

Sometimes I want to hire a skywriting airplane crew. "Barry," says the white smoky cloud, "if you want to compare two numbers, use the double equal sign. Please don't forget to do this." Because I can't afford skywriting, I have to settle for a more modest option: I create a paragraph marked with the Remember icon.

CROSS REFERENCE

"If you don't remember what such-and-such means, see blah-blah-blah," or "For more information, read blahbity-blah-blah."

TRY IT OUT

Writing computer code is an activity, and the best way to learn an activity is to practice it. That's why I've created things for you to try in order to reinforce your knowledge. Many of these are confidence-builders, and some are more challenging. When you first start putting concepts into practice, you'll discover all kinds of issues, quandaries, and roadblocks that didn't occur to you when you started reading about the material. But that's a good thing. Keep at it! Don't become frustrated. Or, if you do become frustrated, visit this book's website (`https://javafordummies.allmycode.com`) for hints and solutions.

This icon calls attention to useful material that you can find online. Check it out!

Occasionally, I run across a technical tidbit. The tidbit may help you understand what the people behind the scenes (the people who developed Java) were thinking. You don't have to read it, but you may find it useful. You may also find the tidbit helpful if you plan to read other (geekier) books about Java.

Beyond the Book

In addition to what you're reading right now, this book comes with a free, access-anywhere Cheat Sheet containing code that you can copy and paste into your own Java program. To get this Cheat Sheet, simply go to `www.dummies.com` and type **Java For Dummies Cheat Sheet** in the Search box.

For your extra reading pleasure, I've also arranged for my publishers to make available for download two appendices and a bonus chapter: The details are as follows:

- **Appendix A: Setting Up Your Computer,** which gives you the skinny on how to prep your computer for your upcoming Java adventure
- **Appendix B: Who's Afraid of Java Documentation?,** a non-nonsense guide to Java's API documentation
- **Bonus Chapter 1: Pat Your Head While You Rub Your Belly**, a guide for those interested in multitasking with your Java code

All three can be found at `www.dummies.com/go/javafd9e`.

Where to Go from Here

If you've gotten this far, you're ready to start reading about Java application development. Think of me (the author) as your guide, your host, your personal assistant. I do everything I can to keep things interesting and, most importantly, to help you understand.

If you like what you read, send me a note. My email address, which I created just for comments and questions about this book, is `JavaForDummies@allmycode.com`. (In case you're wondering, I answer my own emails. I don't use bots or paid assistants.) And don't forget — for the latest updates, visit this book's website. The site's address is `https://javafordummies.allmycode.com`.

1 Getting Started with Java

IN THIS PART . . .

Install the software you need for developing Java programs.

Find out how Java fits into today's technology scene.

Run your first complete Java program.

IN THIS CHAPTER

- What Java is
- Where Java came from
- Why Java is so cool
- Three ways to write computer code

Chapter 1
All about Java

Say what you want about computers. As far as I'm concerned, computers are good for just two simple reasons:

- **When computers do work, they feel no resistance, no stress, no boredom, and no fatigue.** Your computer can work 24/7 making calculations for `www.climateprediction.net` — a distributed computing project to model the world's climate change. Or, have your computer crunch numbers for `Rosetta@home` — a site that models proteins to help cure major illnesses. Will you feel sorry for my computer because it's working so hard? Will the computer complain? No.

  You can make demands, give the computer its orders, and crack the whip. Will you (or should you) feel the least bit guilty? Not at all.

- **Computers move ideas, not paper.** Not long ago, whenever you wanted to send a message to someone, you hired a messenger. The messenger mounted a horse and delivered your message personally. The message was recorded on paper or parchment or a clay tablet or whatever other physical medium was available at the time.

  This whole process seems wasteful now, but that's only because you and I are sitting comfortably in the electronic age. Messages are ideas, and physical objects like ink, paper, and horses have little or nothing to do with real ideas; they're just temporary carriers of ideas (even though people used them for

several centuries to carry ideas). Nevertheless, the ideas themselves are paperless, horseless, and messengerless.

The neat thing about computers is that they carry ideas efficiently. They carry nothing but the ideas, a couple of photons, and some electrical power. They do this with no muss, no fuss, and no extra physical baggage.

When you start dealing efficiently with ideas, something very nice happens: Suddenly, all overhead is gone. Instead of pushing paper and trees, you're pushing numbers and concepts. Without the overhead, you can do things much faster and do things that are far more complex than ever.

What You Can Do with Java

It would be nice if all this complexity were free, but, unfortunately, it isn't. Someone has to think hard and decide exactly what to ask the computer to do. After that thinking takes place, someone has to write a set of instructions for the computer to follow.

Given the current state of affairs, you can't write these instructions in English or any other language that people speak. Science fiction is filled with stories about people who make simple requests of robots and get back disastrous, unexpected results. English and other such languages are unsuitable for communication with computers, for several reasons:

- **An English sentence can be misinterpreted.** "No parking. Violators will be towed away at the owner's expense." So, the parking space's owner will pay to tow my car? That's not so bad!
- **It's difficult to weave a complicated command in English.** "Join flange A to protuberance B, making sure to connect only the outermost lip of flange A to the larger end of the protuberance B while joining the middle and inner lips of flange A to grommet C."
- **English sentences have lots of extra baggage.** "Sentences have unneeded words."
- **English can be difficult to interpret.** "John Wiley & Sons, Inc. shall pay net earnings to the Author ('Barry Burd') or the Author's assigns upon submittal of *Java For Dummies,* 9th Edition ('the Work') minus prepaid and accrued expenses and deferred charges, either directly or indirectly, within the meaning of section 4942(j)(3) or 4965(k)(5) for calendar year 2024."

To tell a computer what to do, you have to use a special language to write terse, unambiguous instructions. A special language of this kind is called a *computer programming language.* A set of instructions written in such a language is called a *program.* When looked at as a big blob, these instructions are called *software* or *code.* Here's what code looks like when it's written in Java:

```
void main() {
    var checkAmount = 1257.63;
    println("Pay to the order of Dr. Barry Burd $" + checkAmount);
}
```

You may argue that ChatGPT and other generative AI tools narrow the gap between informal English and carefully written code. That's true to some extent. But computer code isn't going away anytime soon. Generative AI may hallucinate. And, in some critical applications, hallucinations are unacceptable. Besides, GPT tools don't create themselves. At this very moment (whenever you're reading my book, that is), thousands of people around the world are writing code to make GPT tools faster and more versatile. Programming isn't a dying art. It's a growing endeavor.

Why You Should Use Java

It's time to celebrate! You've just picked up a copy of *Java For Dummies,* 9th Edition, and you're reading Chapter 1. At this rate, you'll be an expert Java programmer* in no time at all, so rejoice in your eventual success by throwing a big party.

To prepare for the party, I'll bake a cake. I'm lazy, so I'll use a ready-to-bake cake mix. Let me see: Add water to the mix and then add butter and eggs — hey, wait! I just looked at the list of ingredients. What's MSG? And what about propylene glycol? That's used in antifreeze, isn't it?

I'll change plans and make the cake from scratch. Sure, it's a little harder, but that way, I get exactly what I want.

Computer programs work the same way: You can use somebody else's program or write your own. If you use somebody else's program, you use whatever you get. When you write your own program, you can tailor the program especially for your needs.

* In professional circles, a developer's responsibilities are usually broader than those of a programmer. But, in this book, I use the terms *programmer* and *developer* almost interchangeably.

Writing computer code is a big, worldwide industry. Companies do it, freelance professionals do it, hobbyists do it — all kinds of people do it. A typical big company has teams, departments, and divisions that write programs for the company. But you can write programs for yourself or for someone else, for a living or for fun. In a recent estimate, the number of lines of code written each day by programmers in the world exceeds the number of methane molecules on the planet Jupiter.** Take almost anything that can be done with a computer — with the right amount of time, you can write your own program to do it. (Of course, the "right amount of time" may be quite long, but that's not the point. Many interesting and useful programs can be written in hours or even minutes.)

Gaining Perspective: Where Java Fits In

Here's a brief history:

- **1954–1957: FORTRAN is developed.**

  FORTRAN was the first modern computer programming language. For scientific programming, FORTRAN is a real racehorse. Year after year, FORTRAN is a leading language among computer programmers throughout the world.

- **1959: Grace Hopper at Remington Rand develops the COBOL programming language.**

  The letter *B* in COBOL stands for *Business,* and business is just what COBOL is all about. The language's primary feature is the processing of one record after another, one customer after another, or one employee after another.

  Within a few years after its initial development, COBOL became the most widely used language for business data processing.

  (Fun fact: I once foolishly tried to write a COBOL program to create musical compositions. Talk about picking the wrong tool for the job! It was like brushing my teeth with a nail file.)

- **1960: John McCarthy writes a paper on his new LISP programming language.**

  LISP sets the stage for a style of programming called *functional programming*. For a few decades, functional programming waits in the background while other programming techniques take center stage.

** I made up this fact all by myself.

- **1967: Scientists at the Norwegian Computing Center develop the Simula 67 language.**

  With Simula 67, object-oriented programming is born.

- **1972: Dennis Ritchie at AT&T Bell Labs develops the C programming language.**

  The "look and feel" that you see in this book's examples comes from the C programming language. Code written in C uses curly braces, `if` statements, `for` statements, and other elements.

  In terms of power, you can use C to solve the same problems that you can solve by using FORTRAN or Java or any other modern programming language. The difference between one programming language and another isn't power — the difference is ease and appropriateness of use. That's where the Java language excels.

- **1986: Bjarne Stroustrup (also at AT&T Bell Labs) develops C++.**

  Unlike its C language ancestor, the C++ language supports object-oriented programming. This support represents a huge step forward. (See the next section in this chapter.)

- **May 23, 1995: Sun Microsystems releases its first official version of the Java programming language.**

  Java improves upon the concepts in C++. Java not only supports object-oriented programming but also *enforces the use of* object-oriented programming.

  Additionally, Java is a great general-purpose programming language. A program written in Java runs seamlessly on all major platforms, including Windows, Macintosh, and Linux. With Java, you can write windowed applications, build and explore databases, control handheld devices, and more. Within five short years, the Java programming language has 2.5 million developers worldwide. (I know — I have a commemorative T-shirt to prove it.)

  By 2001, Java has become the number-one language on the world-famous TIOBE Programming Community Index.

- **2007-2017: Java grows up.**

  For a while, updates to Java come very slowly. With disagreements over the implementation of new features, the gap between Java 6 and Java 7 releases is a whopping five years.

  In the meantime, Oracle Corporation purchases Sun Microsystems — a terrific bargain for only $7.4 billion! Then, in 2014, Oracle releases Java 8 — the first Java version with functional programming capabilities.

  For most of the months between 2001 and 2017, Java has maintained its top position on the TIOBE Index.

» **August 2017: Oracle announces its plan to release new versions of Java every six months.**

The release of Java 9 in September 2017 is followed by the rollout of Java 10 in March 2018. Up next is Java 11 in September 2018.

In September 2023, Java 21 is a *l*ong-*t*erm *s*upport (LTS) release. This means that Oracle promises to keep Java 21 running smoothly at least until September 2028. These LTS releases come every two years, so the next rock-solid, take-no-prisoners version of Java is Java 25 in September 2025.

The new release cycle has injected energy into the evolution of the Java programming language.

Java technology powers applications from companies like Microsoft, Uber, LinkedIn, PayPal, Netflix, Airbnb, Google, eBay, Spotify, TripAdvisor, Intel, Pinterest, and Groupon.* The job search site `Monster.com` says:

> "Java is one of the most popular programming languages in use, so it's no surprise it came in as the No. 1 skill tech companies were looking for. According to Oracle, 3 billion mobile phones run Java, along with 125 million TV devices and 89% of desktop computers in the U.S. Java is everywhere and the demand for strong developers is high."**

Well, I'm impressed.

Three Ways to Write Computer Programs

It's 4 in the morning. I'm dreaming about the history course I failed in high school. The teacher is yelling at me: "You have two days to study for the final exam, but you won't remember to study. You'll forget and feel guilty, guilty, guilty."

Suddenly, the phone rings. I'm awakened abruptly from my deep sleep. (Sure, I disliked dreaming about the history course, but I like being awakened even less.) At first, I drop the telephone on the floor. After fumbling to pick it up, I issue a grumpy, "Hello. Who's this?" A voice answers, "I'm a reporter from the Reuters

---

Sources:

*`https://codegym.cc/groups/posts/771-is-java-still-relevant-what-big-companies-use-it`

**`www.monster.com/career-advice/article/programming-languages-you-should-know`

news agency. I'm writing an article about Java, and I need to know all about the programming language in seven words or less. Can you explain it?"

My mind is too hazy. I can't think. So I say the first thing that comes to my mind and then go back to sleep.

Come morning, I hardly remember the conversation with the reporter. In fact, I don't remember how I answered the question. Did I utter a few obscenities and then go back to sleep?

I put on my robe and rush to my computer. When I visit the Reuters website, I see this enormous headline:

Burd Calls Java "A Great Language in Many Different Ways"

A word you can use to impress people

Today's vocabulary word is the word *paradigm*.

First things first: How do you pronounce the word *paradigm*? The *g* is silent. You say "PAA-ra-dime."

Next, what in the world does *paradigm* mean? I looked up *paradigm* in a few dictionaries, and none of the definitions impressed me much. To my mind, a *paradigm* is a way of thinking about something. For example, people used to think about the sun orbiting the Earth. But, in 1543, Copernicus proposed that the Earth orbits the sun. The Copernican Revolution was a paradigm shift.

Over the years, several computer programming paradigms have emerged. The following sections describe three of them: imperative programming, object-oriented programming, and functional programming.

Imperative programming

Once upon a time, almost every programming language was imperative in nature. With an imperative language, your programs consist mostly of "Do this, then do that" commands. Suppose that you run a small business and you have the three employees listed in Table 1-1.

TABLE 1-1 A Table of Employees

Name	Hours Per Week	Pay
Alice	40	450.00
Bob	20	200.00
Carol	35	300.00

You want each of your employees to work an additional hour every week. In exchange for this, you'll pay each employee $50 more. Here's how you might do it with an imperative style:

```
For each of my employees:
    Get the employee's hours per week,
    add 1 to that number,
    make that the new value of the employee's hours per week.
    Get the employee's current pay,
    add $50 to that amount,
    make that amount the new value of the employee's pay.
```

REMEMBER

What you see here isn't a real computer program. It's an informal, English-language description of the way a real program might work. Since it's not real code, we call it *pseudocode*.

In an *imperative program*, your code is simply a long list of commands. You may set aside some commands to be run separately and give that block of commands a name. But, one way or another, the code's fundamental building blocks are individual commands.

A typical command either gets a stored value or sets a stored value. For example, one command may get an employee's current pay while another command sets the employee's new pay. With imperative programming, each value is loosely connected to all the other values. For example, when your code changes an employee's hours per week, the code happens to change the employee's pay. Other than that, there may be no relationship between hours per week and pay. Each value is an entity on its own.

Object-oriented programming

I often say that imperative programming is "verbs first" and that object-oriented programming (OOP) is "nouns first." With imperative programming, you begin with verbs like *get* and *add.* With object-oriented programming, you begin by defining the kinds of things that your code deals with.

Here's pseudocode to describe the way an object-oriented program defines an employee in your company:

```
An Employee has:
    a name,
    a number of hours per week,
    a pay amount,
    the following code, called add1Hour:
        Add 1 to my hours per week;
    the following code, called add50Dollars:
        Add $50 to my pay.
```

When you write an object-oriented program, you translate this pseudocode into formal Java code and call that code the *Employee class*. Your program may include code for an `Employee` class, a `Customer` class, a `Supplier` class, and many other classes.

Here's something interesting: An `Employee` has data (the employee's name, hours per week, and pay), but an `Employee` also has code for working with the data (`add1Hour` and `add50Dollars`). In object-oriented programming, each class owns code for working with that class. It's as if you'd raise Alice's salary by saying "Alice, please raise your salary." As strange as that may sound, it's a pivotal feature of OOP. The underlying idea is to keep everything you want to know about an `Employee` (data and code) together in one place.

Notice that the `Employee` class code says nothing about any of the individual employees in Table 1-1. The `Employee` class is only a description of the kinds of information that you'll eventually store for each employee. When you code the `Employee` class, you're not creating any actual employees. You can think of the `Employee` class as a table with nothing but a header row. (See Table 1-2.)

TABLE 1-2 A Tabular Representation of the Employee Class

Name	Hours per week	Pay	Add one hour	Add $50 pay

REMEMBER

In object-oriented programming, a class is like a table's header row. A class is the idea behind a certain kind of thing. When I talk about the class of employees, I'm talking about the fact that each employee has a name, a weekly pay amount, a way of adding to the weekly pay, and so on. The pay amounts may be different for different employees, but that doesn't matter. When I talk about a class of things, I'm focusing on the properties that each of the things possesses.

Let's pause for a minute. How do you feel about Table 1-2? Do you like Table 1-2?

Ugh! I don't like Table 1-2. An `Employee` table with no actual employees isn't complete. It's like a day without chocolate. Classes alone don't give object-oriented programming its power.

To complete the picture, assume that you already have Java code that describes the `Employee` class. Now it's time to make use of that code. So you write Java code whose translation into informal English looks something like this:

```
Create an object named Alice from the Employee class. Alice works 40 hours per
   week with pay $450.
Create an object named Bob from the Employee class. Bob works 20 hours per week
   with pay $200.
Create an object named Carol from the Employee class. Carol works 35 hours per
   week with pay $300.
```

You may be uncomfortable calling Alice, Bob, and Carol "objects", but remember that the people themselves aren't objects. It's the information about these people that you call "objects." Since you fashioned each object from the `Employee` class, you call each of these objects an *instance* of the `Employee` class.

REMEMBER

Because Java is an object-oriented programming language, your primary goal is to describe classes and objects. A class is the idea behind a certain kind of thing. An object is a concrete instance of a class. First, you define a class. Then, from the class definition, you construct individual objects.

Having written the code to create three objects, you now have a more complete table. (See Table 1-3.)

TABLE 1-3 The `Employee` Class with Three Objects

Name	Hours per week	Pay	Add one hour	Add $50 pay
Alice	40	450.00	40 + 1 is 41	450 + 50 is 500
Bob	20	200.00	20 + 1 is 21	200 + 50 is 250
Carol	35	300.00	35 + 1 is 36	300 + 50 is 350

Think of the header row in Table 1-3 as the `Employee` class. Then each of the other rows is an object – an instance of the `Employee` class.

Now that you've created three `Employee` instances, you can write code to increase each employee's hour and pay. Here's the pseudocode version:

```
For each employee,
    run that employee's add1Hour code,
    run that employee's add50Dollars code.
```

Once you've created a class and its objects, the actual work of changing your employees' hours and pay is straightforward and concise. And that, my friends, is what object-oriented programming is all about.

Wait! What do I hear you saying to yourself? You say this description of object-oriented programming is vague. You don't really understand it yet. Okay. I admit it. I can't teach you all about OOP in Chapter 1. There's a lot more detail in the chapters to come. Just remember that Java is, first and foremost, an object-oriented programming language. In Java, everything is part of the object-oriented framework. To my mind, Java's implementation of the object-oriented paradigm is the best there is.

TIP

With a name like "object-oriented programming," you'd expect the word "object" to appear several times on each page of this book. But that's not the case. Each object that you create with the `Employee` class is an "*instance* of the `Employee` class," also known as an "`Employee` class *instance*" or just "`Employee` *instance*." If you use the term "`Employee` object" instead of "`Employee` instance," people will know what you mean. And sometimes, you can say "an `Employee`" when you really mean "an `Employee` instance." But your best bet is to remember that each object you construct using the `Employee` class is called an `Employee` instance.

Functional programming

Functional programming has several moving parts. But, in this chapter, you can think of functional programming as an assembly line where data travels from one station to another. Each station makes a new bundle of data from the old bundle and sends the new bundle on its way. One station, usually called `forEach` or `map`, takes a bunch of things along with a piece of code and applies the code to each of the things.

You can use `forEach` to increase the hours and pay for all your employees. Here's some pseudocode to show you how it works:

```
Start with the list of employees,
then apply forEach with the code add1Hour to the list,
then apply forEach with the code add50Dollars to the list.
```

By the way, as this chapter has progressed, I've been hearing from Alice. Table 1-3 says she'll have to work more than 40 hours a week, and Alice is upset about that. Why don't you change your plans? Consider adding an hour to each employee's work week but do this only for employees who aren't already working 40 hours.

Functional programming has something called `filter`, which accepts only items that satisfy certain criteria. Here's how `filter` can keep Alice from working overtime:

```
Start with the list of employees,
then filter for employees who currently work less than 40 hours,
then apply forEach with the code add1Hour to the filtered list,
then apply forEach with the code add50Dollars to the filtered list.
```

But wait! When you keep Alice's hours at 40, will your employees be working enough hours to keep your business afloat? To find out, you pull another functional programming trick out of your hat. This trick is called `reduce`. It takes a piece of code and combines a list's values using that code. Here's what you get:

```
Start with the list of employees,
then filter for employees who currently work less than 40 hours,
then apply forEach with the code add1Hour to the filtered list,
then apply forEach with the code add50Dollars to the filtered list,
then apply forEach with code to retrieve hours per week from the filtered list,
then apply reduce with the code to do addition to the filtered list of
   employees' hours per week.
```

The diagram in Figure 1-1 shows how this functional programming strategy resembles an assembly line.

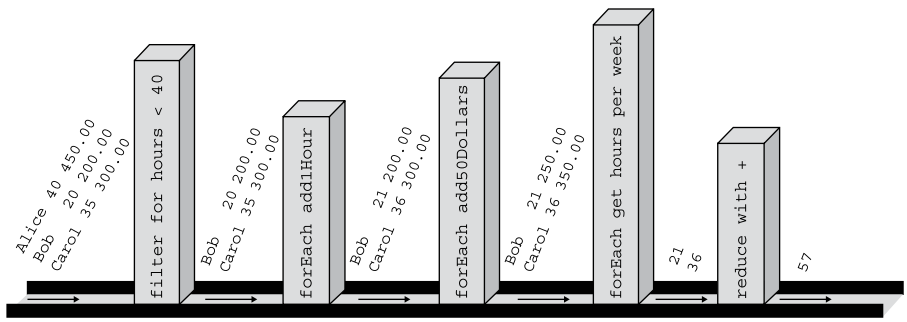


FIGURE 1-1: The functional programming assembly line.

Like my rambling about OOP in the previous section, this section's description of functional programming is decidedly vague. But that's okay because Chapter 12 has complete examples to show how functional programming works in Java.

What's Next?

This chapter is filled with general descriptions of things. A general description is good when you're just getting started, but you don't truly understand things until you get to know some specific info, as laid out in the next several chapters.

So please, turn the page. The next chapter can't wait for you to read it.

IN THIS CHAPTER

- Installing software development tools
- Peeking inside the tools
- Writing and running your own Java programs

Chapter 2
All about Software

The best way to get to know Java is to do Java. When you're doing Java, you're writing, testing, and running your own Java programs. This chapter describes the kind of software you use for all those Java-related tasks. The chapter has *general* instructions to help you set up your computer, but it has no detailed instructions. If you want detailed instructions, visit this book's website at `www.dummies.com/go/javafd9e`.

Get Ready for Java

If you're a seasoned veteran of computers and computing (whatever that means) and you're too jumpy to follow the detailed instructions on this book's website, you can try installing the required software by reading these general instructions:

1. **Install the newest release of the Java Development Kit (JDK).**

   A Java Development Kit is a bunch of software that makes all Java programs work.

   To install a Java Development Kit, visit `www.oracle.com/java/technologies/downloads` and follow that website's instructions. Get the newest available version of Java. To run this book's examples, I used Java 23. But if you're reading this book in 2028 and you can get Java 30, go for it!

But wait! What does it mean to "make all Java programs work"? I answer that question later in this chapter, in the section "The Inside Scoop."

2. **Install an integrated development environment.**

An *integrated development environment (IDE)* is a program to help you compose and test new software. It's like a glorified version of Microsoft Word for writing computer code. For this book's examples, you can use almost any IDE that supports Java.

Here's a list of IDEs that are most popular among professional developers:

- **IntelliJ IDEA** (`www.jetbrains.com/idea`)
- **Eclipse** (`www.eclipse.org/downloads`)
- **Visual Studio Code, also known as VS Code** (`https://code.visualstudio.com`)
- **NetBeans** (`https://netbeans.apache.org`)

Some IDEs are made especially for students, educators, and other specialized communities. These include BlueJ, DrJava, Greenfoot, JCreator, jGrasp, and several others.

TECHNICAL STUFF

If you like roughing it, you can write and run Java programs without using an IDE: Just type your Java program in a plain-text editor (such as Windows Notepad) and run the program on your operating system's command line (Windows MS-DOS, macOS Terminal, or whatever). It's not fun to develop software this way, but it makes you feel like a big shot.

This book's website has detailed instructions for installing and using the most commonly used IDEs.

3. **Test your installed software.**

What you do in this step depends on which IDE you choose in Step 2. Anyway, here are some general instructions:

a. Launch your IDE (Eclipse, IntelliJ IDEA, NetBeans, or whatever).

b. In the IDE, create a new Java project.

c. Within the Java project, create a new Java class named `Main`. (Choosing File ➪ New ➪ Java Class works in most IDEs.)

d. Edit the new `Main.java` file by typing the following lines of code:

```
public class Main {
    public static void main(String[] args) {
        System.out.print(12345);
    }
}
```

For most IDEs, you add the code into a big (mostly blank) Editor pane. Try to type the code exactly as you see it here. If you see an uppercase letter, type an uppercase letter. Do the same with all lowercase letters.

What? You say you don't want to type a bunch of code from the book? Well, all right then! Visit this book's website (`http://JavaForDummies.allmycode.com`) to download all the code examples and copy them into the IDE of your choice.

e. Run `Main.java` and make sure that the run's output reads 12345.

TIP

You may find variations on the picture that I paint in the preceding steps. For example, some IDEs come with options for you to install a JDK. In those cases, you can skip Step 1 and march straight to Step 2. Nevertheless, the picture that I paint with the preceding steps is useful and reliable. When you follow my instructions, you might end up with two copies of Java, or two IDEs, but that's okay. You never know when you'll need a spare.

That's it! But remember: Not everyone (computer geek or not) can follow these skeletal instructions flawlessly. So, if you want more details, visit `http://JavaForDummies.allmycode.com`.

TIMING IS EVERYTHING

In 1955, you could buy a hamburger at McDonald's for 15 cents US. You'd drive there in your new car, which cost you about $2,000. To fill your car's tank, you'd pay about 30 cents per gallon. Things change a lot over time, and Java is no exception.

As of September 2024, some of the features in this book's examples are *preview features* in Java 23. A preview feature is one that

- May not work the same way in later versions of Java
- Takes a little bit of extra work to get running

The `Main.java` program I have you create in this chapter works on all Java versions for all time, but the examples in Chapter 3 require Java 23 or later. What's more, you must take special steps to tell the computer that you want to use these preview features. For example, if you're using the IntelliJ IDEA to run your code, you must do the following:

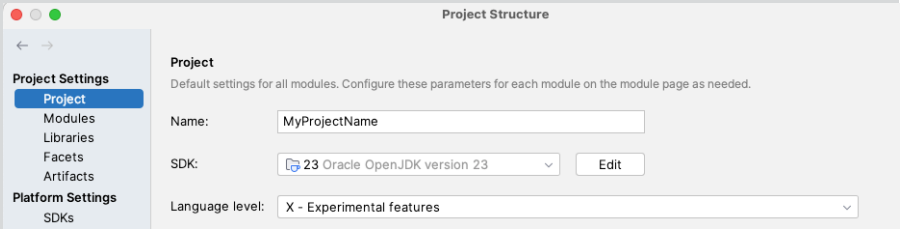
1. Create a project to contain your Java code.
2. On the IntelliJ IDEA main menu, choose File ➪ Project Structure.

(continued)

(continued)

3. On the left side of the resulting Project Structure dialog box, select Project. (See the sidebar figure.)
4. In the dialog box's SDK field, make sure the version number is 23 or higher.
5. In the Language Level field, select Experimental Features. (Again, see the figure.)

If this list of instructions doesn't work, or if you have some other kind of trouble getting Java programs to run, send me an email. My address is `JavaForDummies@allmycode.com`.



The Inside Scoop

One of my acquaintances is a tool-and-die maker. She uses tools to make tools (and dies). I once asked, "Who makes the tools that you use to make tools?" After ignoring her smart-aleck answer, I guessed that a tool-and-die-toolmaker makes tools for tool-and-die makers so that tool-and-die makers can make tools.

A computer programmer does the same kind of thing: A programmer uses existing programs as tools to create new programs. The existing programs and new programs might perform very different kinds of tasks. For example, a Java program (a program you create) might keep track of a business's customers. To create that customer-tracking program, you use several programs belonging to a Java Development Kit. With a JDK's programs, you can create many other useful programs — customer-tracking programs, weather-predicting programs, gaming programs, or programs that run on your mobile phone.

A JDK contains at least 30 different programs — many more, if you count compressed archives and other such items. This section deals with two of the most important JDK components:

- A *compiler* is a program that takes your Java code and turns that code into a bunch of instructions called *bytecode*.

  Humans can't readily compose or decipher bytecode instructions. But certain software that you run on your computer can interpret and carry out bytecode instructions.

- A *Java virtual machine (JVM)* is a program that interprets and carries out bytecode instructions.

The rest of this section describes compilers and Java virtual machines.

What is a compiler?

A compiler is a program that takes your Java code and turns that code into a bunch of instructions called bytecode.

—Barry Burd, *Java For Dummies, 9th Edition*

You're a human being. (Sure, every rule has exceptions. But if you're reading this book, you're probably human.) Anyway, humans can write and comprehend the code in Listing 2-1.

LISTING 2-1: Looking for a Vacant Room

```
// This is part of a Java program.
// It's not a complete Java program.
roomNum = 1;
while (roomNum < 100) {
    if (guests[roomNum] == 0) {
        println("Room " + roomNum + " is available.");
        exit(0);
    } else {
        roomNum++;
    }
}
println("No vacancy");
```

The Java code in Listing 2-1 checks for vacancies in a small hotel (a hotel with room numbers from 1 to 99). You can't run the code in Listing 2-1 without adding several lines. But here in Chapter 2, those additional lines aren't important. What's

important is that, by staring at the code, squinting a bit, and looking past all the code's strange punctuation, you can see what the code is trying to do:

```
Set the room number to 1.
As long as the room number is less than 100,
    Check the number of guests in the room.
    If the number of guests in the room is 0, then
        report that the room is available,
        and stop.
    Otherwise,
        prepare to check the next room by
        adding 1 to the room number.
If you get to the nonexistent room number 100, then
    report that there are no vacancies.
```

If you see no similarities between Listing 2-1 and its English equivalent, don't worry: You're reading *Java For Dummies*, 9th Edition, and, like most human beings, you can learn to read and write the code in Listing 2-1. The code in Listing 2-1 is called *Java source code.*

Here's the catch: Computers aren't human beings. Computers don't normally follow instructions like the instructions in Listing 2-1. That is, computers don't follow Java source code instructions. Instead, computers follow cryptic instructions like the ones in Listing 2-2.

LISTING 2-2: **A Translation of Listing 2-1 into Java Bytecode**

```
aload_0
iconst_1
putfield Hotel/roomNum I
goto 32
aload_0
getfield Hotel/guests [I
aload_0
getfield Hotel/roomNum I
iaload
ifne 26
getstatic java/lang/System/out Ljava/io/PrintStream;
new java/lang/StringBuilder
dup
ldc "Room "
invokespecial java/lang/StringBuilder/<init>(Ljava/lang/String;)V
aload_0
```

```
getfield Hotel/roomNum I
invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;
ldc " is available."
invokevirtual
  java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
iconst_0
invokestatic java/lang/System/exit(I)V
goto 32
aload_0
dup
getfield Hotel/roomNum I
iconst_1
iadd
putfield Hotel/roomNum I
aload_0
getfield Hotel/roomNum I
bipush 100
if_icmplt 5
getstatic java/lang/System/out Ljava/io/PrintStream;
ldc "No vacancy"
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
return
```

The instructions in Listing 2-2 aren't Java source code instructions. They're *Java bytecode* instructions. When you write a Java program, you write source code instructions (like the instructions in Listing 2-1). After you finish writing the source code, your IDE runs a program (a translation tool) on your source code. The program is a *compiler.* The compiler translates your source code instructions into Java bytecode instructions. In other words, the compiler takes code that you can write and understand (like the code in Listing 2-1) and translates it into code that a computer has a fighting chance of carrying out (like the code in Listing 2-2).

You might put your source code in a file named `Hotel.java`. If so, the compiler probably puts the Java bytecode in another file named `Hotel.class`. Normally, you don't bother looking at the bytecode in the `Hotel.class` file. In fact, the compiler doesn't encode the `Hotel.class` file as ordinary text, so you can't examine the bytecode with an ordinary editor. If you try to open `Hotel.class` with Notepad, TextEdit, KWrite, or even Microsoft Word, you see nothing but dots, squiggles, and other gobbledygook. To create Listing 2-2, I had to apply yet another tool to my `Hotel.class` file. That tool displays a text-like version of a Java bytecode file. I used Ando Saabas's Java Bytecode Editor (`https://set.ee/jbe`).

REMEMBER

No one (except for a few crazy programmers in some isolated labs in faraway places) writes Java bytecode. You run software (a compiler) to create Java bytecode. The only reason to look at Listing 2-2 is to understand what a hard worker your computer is.

What is a Java virtual machine?

A Java virtual machine (JVM) is a program that interprets and carries out bytecode instructions.

—Barry Burd, *Java For Dummies, 9th Edition*

In the preceding "What is a compiler?" section, I make a big fuss about computers following instructions like the ones in Listing 2-2. As fusses go, it's a very nice fuss. But if you don't read every fussy word, you may be misguided. The exact wording is " . . . computers follow cryptic instructions *like* the ones in Listing 2-2." The instructions in Listing 2-2 are a lot like instructions that a computer can execute, but generally, computers don't execute Java bytecode instructions. Instead, each kind of computer processor has its own set of executable instructions, and each computer operating system uses the processor's instructions in a slightly different way.

Here's a hypothetical situation: The year is 1992 (a few years before Java was made public) and you run the Linux operating system on a computer that has an old Pentium processor. Your friend runs Linux on a computer with a different kind of processor — a PowerPC processor. (In the 1990s, Intel Corporation made Pentium processors, and IBM made PowerPC processors.)

Listing 2-3 contains a set of instructions to display `Hello world!` on the computer screen.* The instructions work on a Pentium processor running the Linux operating system.

LISTING 2-3: A Simple Program for a Pentium Processor

```
.data
msg:
        .ascii      "Hello,   world!\n"
        len = .     - msg
.text
    .global _start
```

* I paraphrase these Intel instructions from Konstantin Boldyshev's Linux Assembly HOWTO document (`http://tldp.org/HOWTO/Assembly-HOWTO/hello.html`).

```
_start:
        movl $len,%edx
        movl $msg,%ecx5
        movl $1,%ebx
        movl $4,%eax
        int $0x80


        movl $0,%ebx
        movl $1,%eax
        int $0x80
```

Listing 2-4 contains another set of instructions to display `Hello world!` on the screen.** The instructions in Listing 2-4 work on a PowerPC processor running Linux.

LISTING 2-4: **A Simple Program for a PowerPC Processor**

```
.data
msg:
        .string "Hello, world!\n"
        len = . - msg
.text
        .global _start
_start:
        li 0,4
        li 3,1
        lis 4,msg@ha
        addi 4,4,msg@l
        li 5,len
        sc


        li 0,1
        li 3,1
        sc
```

** I paraphrase the PowerPC code from Hollis Blanchard's PowerPC Assembly (`www.scribd.com/document/655168531/PowerPC-assembly-overview`). Hollis also reviewed and critiqued this "What is a Java virtual machine?" section for me. Thank you, Hollis.

The instructions in Listing 2-3 run smoothly on a Pentium processor. But these instructions mean nothing to a PowerPC processor. Likewise, the instructions in Listing 2-4 run nicely on a PowerPC, but these same instructions are complete gibberish to a computer with a Pentium processor. So your friend's PowerPC software might not be available on your computer. And your Intel computer's software might not even run on your friend's computer.

Now go to your cousin's house. Your cousin's computer has a Pentium processor (just like yours), but your cousin's computer runs Windows instead of Linux. What does your cousin's computer do when you feed it the Pentium code in Listing 2-3? It screams, "Not a valid Win32 application" or "Windows can't open this file." What a mess!

Java bytecode creates order from all this chaos. Unlike the code in Listings 2-3 and 2-4, Java bytecode isn't specific to one kind of processor or to a single operating system. Instead, any kind of computer can have a Java virtual machine, and Java bytecode instructions run on any computer's Java virtual machine. The JVM that runs on a Pentium with Linux translates Java bytecode instructions into the kind of code you see in Listing 2-3. And the JVM that runs on a PowerPC with Linux translates Java bytecode instructions into the kind of code you see in Listing 2-4. The same kind of translation takes place for modern processors, like the Intel i9 and Apple Silicon.

If you write a Java program and compile that Java program into bytecode, then the JVM on your computer can run the bytecode, the JVM on your friend's computer can run the bytecode, and the JVM on your grandmother's supercomputer can run the bytecode.

CROSS REFERENCE

For a look at some Java bytecode, see Listing 2-2. ***Remember:*** You never have to write or decipher Java bytecode. Writing bytecode is the compiler's job. Deciphering bytecode is the Java virtual machine's job.

With Java, you can take a bytecode file that you created with a Windows computer, copy the bytecode to who-knows-what kind of computer, and then run the bytecode with no trouble. That's one of the many reasons Java has become popular so quickly. This outstanding feature, which gives you the ability to run code on many different kinds of computers, is called *portability.*

What makes Java bytecode so versatile? This fantastic universality enjoyed by Java bytecode programs comes from the Java virtual machine. The Java virtual machine is one of the tools that you *must* have on your computer. (Refer to this chapter's "Get Ready for Java" section.)

Imagine that you're the Windows representative to the United Nations Security Council. (See Figure 2-1.) The Macintosh representative is seated to your right,

and the Linux representative is to your left. (Naturally, you don't get along with either of these people. You're always cordial to one another, but you're never sincere. What do you expect? It's politics!) The distinguished representative from Java is at the podium. The Java representative is speaking in bytecode, and neither you nor your fellow ambassadors (Mac and Linux) understand a word of Java bytecode.

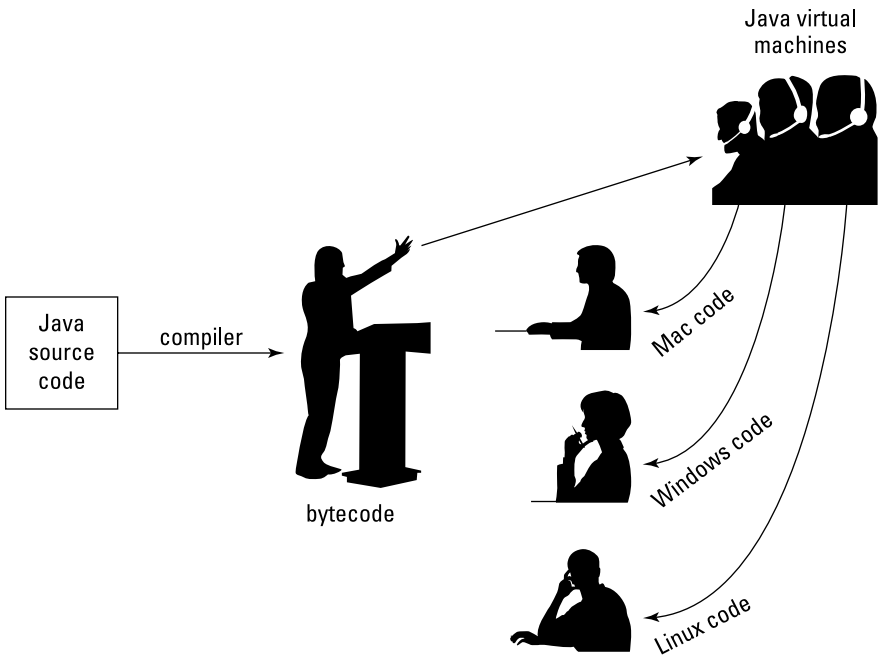


FIGURE 2-1: An imaginary meeting of the UN Security Council.

But each of you has an interpreter. Your interpreter translates from bytecode to Windows while the Java representative speaks. Another interpreter translates from bytecode to Macintosh-ese. And a third interpreter translates bytecode into Linux-speak.

Think of your interpreter as a virtual ambassador. The interpreter doesn't really represent your country, but the interpreter performs one of the important tasks that a real ambassador performs. The interpreter listens to bytecode on your behalf. The interpreter does what you would do if your native language were Java bytecode. The interpreter pretends to be the Windows ambassador and sits through the boring bytecode speech, taking in every word and processing each word in some way or another.

You have an interpreter — a virtual ambassador. In the same way, a Windows computer runs its own bytecode-interpreting software. That software is the Java virtual machine.

A Java virtual machine is a proxy, an errand boy, a go-between. The JVM serves as an interpreter between Java's run-anywhere bytecode and your computer's own system. While it runs, the JVM walks your computer through the execution of bytecode instructions. The JVM examines your bytecode, bit by bit, and carries out the instructions described in the bytecode. The JVM interprets bytecode for your Windows system, your Mac, or your Linux box, or for whatever kind of computer you're using. That's a good thing. It's what makes Java programs more portable than programs in any other language.

Developing Software

All this has happened before, and it will all happen again.

—Peter Pan *(J. M. Barrie) and* Battlestar Galactica *(2003–2009, NBC Universal)*

When you create a Java program, you repeat the same steps over and over again. Figure 2-2 illustrates the cycle.

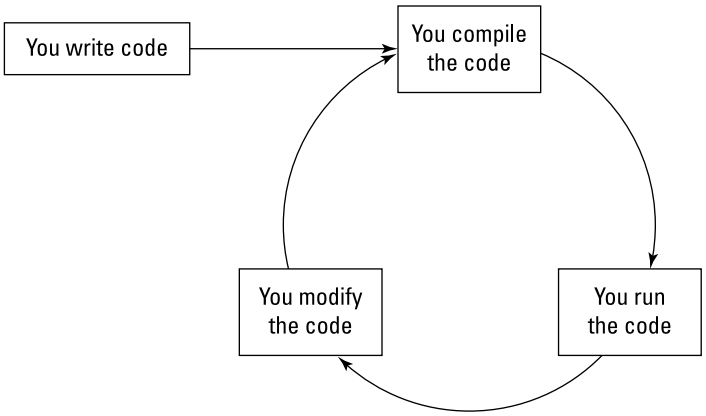


FIGURE 2-2: Developing a Java program.

First, you write a program. After writing the first draft, you repeatedly compile, run, and modify the program. With a little experience, the compile and run steps become easy to carry out. In many cases, one mouse-click starts the compilation or the run.

However, writing the first draft and modifying the code are not one-click tasks. Developing code requires time and concentration.

REMEMBER

Never be discouraged when the first draft of your code doesn't work. For that matter, never be discouraged when the 25th draft of your code doesn't work. Rewriting code is one of the most important things you can do (aside from ensuring world peace).

When people talk about writing programs, they use the wording in Figure 2-2. They say, "You compile the code" and "You run the code." But the "you" isn't always accurate, and the "code" differs slightly from one part of the cycle to the next. Figure 2-3 describes the cycle from Figure 2-2 in a bit more detail.

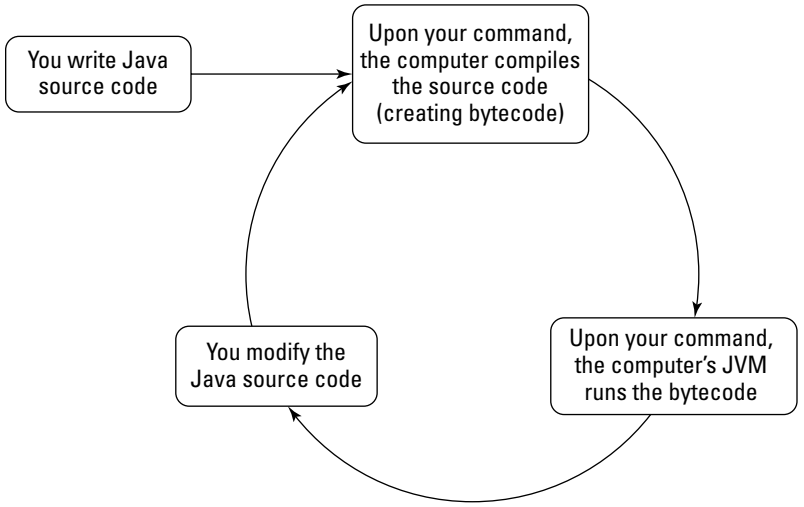

You write Java source code
Upon your command, the computer compiles the source code (creating bytecode)
Upon your command, the computer's JVM runs the bytecode
You modify the Java source code

FIGURE 2-3: Who does what with which code?

TIP

For most people's needs, Figure 2-3 contains too much information. If I click a Run icon, I don't have to remember that the computer runs code on my behalf. And, for all I care, the computer can run my original Java code or some bytecode knockoff of my original Java code. In fact, many times in this book, I casually write "when you run your Java code," or "when the computer runs your Java program." You can live a very happy life without looking at Figure 2-3. The only use for Figure 2-3 is to help you if the loose wording in Figure 2-2 confuses you. If Figure 2-2 doesn't confuse you, ignore Figure 2-3.

Spoiler Alert!

In the next chapter, you start running Java code. That's what you really want from this book. Isn't it?

IN THIS CHAPTER

- Speaking the Java language: The API and the language specification
- Taking a first glance at Java code
- Understanding the parts of a simple program
- Documenting your code

Chapter 3
Using the Basic Building Blocks

"Все мысли, которые имеют огромные последствия всегда просты. (All great ideas are simple.)"

—Leo Tolstoy

This quotation applies to all kinds of things — things like life, love, and computer programming. That's why this chapter takes a multilayered approach. In this chapter, you get your first details about Java programming. And in discovering details, you'll see the simplicities.

Speaking the Java Language

If you try to picture in your mind the entire English language, what do you see? Maybe you see words, words, words. (That's what Hamlet saw.) Looking at the language under a microscope, you see one word after another. The bunch-of-words image is fine, but if you step back a bit, you may see two other things:

- The language's grammar
- Thousands of expressions, sayings, idioms, and historical names

The first category (the grammar) includes rules like this: "The verb agrees with the noun in number and person." The second category (expressions, sayings, and stuff) includes knowledge like this: "Julius Caesar was a famous Roman emperor, so don't name your son Julius Caesar, unless you want him to get beaten up every day after school."

The Java programming language has all the aspects of a spoken language like English. Java has words, grammar, commonly used names, stylistic idioms, and other such elements.

The grammar and the common names

The people at Sun Microsystems who created Java thought of Java as having separate parts. Just as English has its grammar and commonly used names, the Java programming language has its specification (its grammar) and its application programming interface (its commonly used names). Along with these parts come two important documents:

- **The Java Language Specification:** This documentation includes rules like this: "Always put an open parenthesis after the word *for*" and "Use an asterisk to multiply two numbers."
- **The API Specification:** Java's *application programming interface (API)* contains thousands of names that were added to Java after the language's grammar was defined. These names range from the commonplace to the exotic. For example, one name — the name *JFrame* — represents a window on your computer's screen. A more razzle-dazzle name — *pow* — helps you raise 5 to the tenth power or raise whatever to the whatever-else power. Other names help you listen for the user's button clicks, query databases, and do all kinds of useful things.

You can download the language specification by poking around at `https://docs.oracle.com/javase/spec`, but I don't recommend doing it. With the language spec, you can settle subtle arguments about edge cases in the behavior of Java programs. But the spec is far too detailed for the day-to-day study of Java.

The second document — the API Specification — is the go-to document for most of your Java programming needs. The API contains thousands and thousands of names and keeps growing with each new Java language release. That may seem scary, but there's good news — you don't have to memorize anything in the API. Nothing. None of it. You can look up the stuff you need to use in the documentation and ignore the stuff you don't need. What you use often, you'll remember. What you don't use often, you'll forget (like any other programmer).

The API document for Java 23 lives online at `https://docs.oracle.com/en/java/javase/23/docs/api/index.html`. For later releases of Java, change *23* in the URL to the appropriate version number.

REMEMBER

No one knows all there is to know about the Java API. If you're a Java programmer who frequently writes programs that open new windows, you know how to use the API's `JFrame` class. If you seldom write programs that open windows, the first few times you need to create a window, you can look up the `JFrame` class in the API documentation. My guess is that if you prevented a typical Java programmer from looking up anything in the API documentation, the programmer would be able to use less than 2 percent of all names in the Java API.

You may love the *For Dummies* style, but unfortunately, Java's official API documentation isn't written that way. The API documentation is both concise and precise. For some help in deciphering the API documentation's language and style, visit this book's website at `www.dummies.com/go/javafd9e`. (The link to this content is also available at my `https://JavaForDummies.allmycode.com` site.)

In a way, nothing about the Java API is special. Whenever you write a Java program — even the smallest, simplest Java program — you create a class that's on par with any of the classes defined in the official Java API. The API is just a set of classes and other names that were created by ordinary programmers who happen to participate in the official Java Community Process (JCP). Unlike the names you create, the names in the API are distributed with every version of Java. (I'm assuming that you, the reader, are not a participant in the Java Community Process. But, with a fine book like *Java For Dummies,* 9th Edition, one never knows.)

If you're interested in the JCP's activities, visit `www.jcp.org`.

The folks at the JCP don't keep the Java programs in the official Java API a secret. If you want, you can look at all these programs by visiting `https://github.com/openjdk/jdk/tree/master/src/java.base/share/classes/java`.

The words in a Java program

A hard-core Javateer will say that the Java programming language has four kinds of words: keywords, restricted keywords, literals, and identifiers. This is true. But the bare truth, with no other explanation, isn't useful. So I dress up the truth a bit by thinking in terms of three kinds of words: keywords, identifiers that ordinary programmers like you and me create, and identifiers from the API.

The differences among these three kinds of words are similar to the differences among words in the English language. In the sentence "Sam is a person," the word *person* is like a Java keyword. No matter who uses the word *person,* the word

always means roughly the same thing. (Sure, you can think of bizarre exceptions in English usage, but please don't.)

The word *Sam* is like a Java identifier because Sam is a name for a particular person. Words like *Sam*, *Dinswald*, and *McGillimaroo* aren't prepacked with meaning in the English language. These words apply to different people, depending on the context, and become names when parents pick one for their newborn kid.

Now consider the sentence "Julius Caesar is a person." If you utter this sentence, you're probably talking about the fellow who ruled Rome until the Ides of March. Although the name Julius Caesar isn't hardwired into the English language, almost everyone uses the name to refer to the same person. If English were a programming language, the name Julius Caesar would be an API identifier.

Here's how I, in my mind, divide the words in a Java program into categories:

- **Keywords:** A *keyword* is a word that has its own special meaning in the Java programming language, and that meaning doesn't change from one program to another. Examples of keywords in Java are `if`, `else`, and `do`.

  The JCP committee members, who have the final say on what constitutes a Java program, have chosen all the Java keywords. If you think about the two parts of Java, which I discuss earlier, in the section "The grammar and the common names," the Java keywords belong solidly to the language specification.

- **Identifiers:** An *identifier* is a name for something. The identifier's meaning can change from one program to another, but some identifiers' meanings tend to change more:

  - *Identifiers created by you and me:* As a Java programmer (yes, even as a novice Java programmer), you create new names for classes and other items you describe in your programs. Of course, you may name something `Prime`, and your coworker writing code two cubicles down the hall can name something else `Prime`. That's okay because Java has no predetermined meaning for `Prime`. In your program, you can make `Prime` stand for the Federal Reserve's prime rate. And your friend down the hall can make `Prime` stand for the "bread, roll, preserves, and prime rib." No conflict arises, because you and your coworker are writing two different Java programs.

  - *Identifiers from the API:* The JCP members have created names for many things and thrown tens of thousands of these names into the Java API. The API comes with each version of Java, so these names are available to anyone who writes a Java program. Examples of such names are `String`, `Integer`, `JWindow`, `JButton`, `JTextField`, and `File`.

Strictly speaking, the meanings of the identifiers in the Java API aren't cast in stone. Although you can make up your own meanings for `JButton` or `JWindow`, this isn't a good idea. If you did, you would confuse the dickens out of other programmers, who are used to the standard API meanings for these familiar identifier names. But even worse, when your code assigns a new meaning to an identifier like `JButton`, you lose any computational power that was created for the identifier in the API code. The programmers at Sun Microsystems, Oracle, and the Java Community Process did all the work of writing Java code to handle buttons. If you assign your own meaning to `JButton`, you're turning your back on all the progress made in creating the API.

To see the list of Java keywords, go to `www.dummies.com` and enter *Java For Dummies cheat sheet* in the Search box.

Checking Out Java Code for the First Time

The first time you look at somebody else's Java program, you may tend to feel a bit queasy. The realization that you don't understand something (or many things) in the code can make you nervous. I've written hundreds (maybe thousands) of Java programs, but I still feel insecure when I start reading someone else's code.

The truth is that finding out about a Java program is a bootstrapping experience. First, you gawk in awe of the program. Then you run the program to see what it does. Then you stare at the program for a while or read someone's explanation of the program and its parts. Then you gawk a little more and run the program again. Eventually, you come to terms with the program. (Don't believe the wise guys who say they never go through these steps. Even the experienced programmers approach a new project slowly and carefully.)

In Listing 3-1, you get a blast of Java code. (Like all novice programmers, you're expected to gawk humbly at the code.) Hidden in the code, I've placed some important ideas, which I explain in detail in the next section. These ideas include the use of methods and Java statements.

LISTING 3-1: Two Plus Two

```
void main() {
    println(2 + 2);
}
```

WARNING

tHE jAVA PROGRAMMING LANGUAGE IS cASe-sEnsITiVE. If you change a lowercase letter in a word to an UpperCase letter, you can change the word's meaning. cHANGING case can make the entire word go from being meaningful to being meaningless. In the first line of Listing 3-1, you can't replace void with Void or println with printLn. iF YOU DO, THE WHOLE PROGRAM STOPS WORKING.

REMEMBER

In `println`, the next-to-last character is a lowercase letter *l*, not a digit *1*.

You don't have to type the code in Listing 3-1 (or in any of this book's listings). To download all the code in this book, visit the book's website (`http://JavaForDummies.allmycode.com`).

When you run the program from Listing 3-1, the computer displays the number `4`. (In case you don't believe me, see Figure 3-1. In that figure, ignore everything except the number 4.) Now, I admit that writing and running a Java program is a lot of work just to find out that `2 + 2` equals 4, but every endeavor has to start somewhere.

A TRIED-AND-TRUE JAVA PROGRAM

The code in Listing 3-1 uses Java 23 preview features. As of September 2024, these features are available in Java, but they're not yet officially part of Java. So, when you try to run this example, your computer may complain to you. Chapter 2 has some hints on dealing with such complaints. But if you'd rather jettison the preview features, you can change Listing 3-1 as follows:

```
class Main {
    public static void main(String[] args) {
        System.out.println(2 + 2);
    }
}
```

You can do this with each example in Chapters 3 through 6 of this book. Just follow these instructions:

1. Add `class Main {` at the top of the program.
2. Add a closed curly brace `}` at the bottom of the program.
3. Change `void main()` to `public static void main(String[] args)`.
4. Change `println` to `System.out.println`.

A few examples in Chapters 3 through 6 require an additional tweak. Look for hints surrounding each code listing.

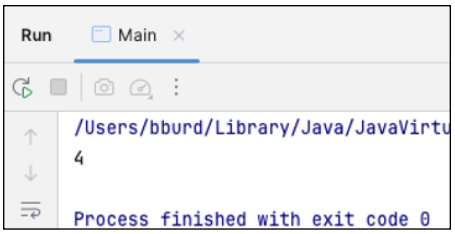


FIGURE 3-1: Running the program in Listing 3-1.

This book's website (`http://JavaForDummies.allmycode.com`) has instructions to help you run Java programs such as the code in Listing 3-1.

In the following section, you do more than just admire the program's output. After you read the following section, you will understand what makes the program in Listing 3-1 work.

Understanding a Simple Java Program

This section presents, explains, analyzes, dissects, and otherwise demystifies the Java program shown previously in Listing 3-1.

The Java method

You're working as an auto mechanic in an upscale garage. Your boss, who's always in a hurry and has a habit of running words together, says, "fixTheAlternator on that junkyOldFord." Mentally, you run through a list of tasks: "Drive the car into the bay, lift the hood, find a wrench, loosen the alternator belt," and so on. Three things are going on here:

- **You have a name for what you're supposed to do.** The name is *fixTheAlternator.*
- **In your mind, you have a list of tasks associated with the name *fixTheAlternator.*** The list includes "Drive the car into the bay, lift the hood, find a wrench, loosen the alternator belt," and so on.
- **You have a grumpy boss who's telling you to do all this work.** Your boss gets you working by saying, "fixTheAlternator." In other words, your boss gets you working by saying the name of the task you're supposed to do.

In this scenario, using the word *method* wouldn't be a big stretch. You have a method for doing something with an alternator. Your boss calls that method into

action, and you respond by doing all the things in the list of instructions that you associate with the method.

If you believe all that (and I hope you do), you're ready to read about Java methods. In Java, a *method* is a list of things to do. Every method has a name, and you tell the computer to do the things in the list by using the method's name in your program.

I've never written a program to get a robot to fix an alternator. But, if I did, the program might include a `fixTheAlternator` method. The list of instructions in my `fixTheAlternator` method would look something like the text in Listing 3-2.

WARNING

Don't scrutinize Listings 3-2 and 3-3 too carefully — all the code in them is fake! I made up this code so that it looks a lot like real Java code, but it's not real. What's more important, the code in Listings 3-2 and 3-3 isn't meant to illustrate all the rules about Java. So, if you have a grain of salt handy, take it with Listings 3-2 and 3-3.

LISTING 3-2: A Method Declaration

```
void fixTheAlternator(onACertainCar) {
   driveInto(car, bay);
   lift(hood);
   get(wrench);
   loosen(alternatorBelt);
   ...
}
```

Somewhere else in my Java code (somewhere outside of Listing 3-2), I need an instruction to call my `fixTheAlternator` method into action. The instruction to call the `fixTheAlternator` method into action may look like the line in Listing 3-3.

LISTING 3-3: A Method Call

```
fixTheAlternator(junkyOldFord);
```

Now that you have a basic understanding of what a method is and how it works, you can dig a little deeper into some useful terminology:

- If I'm being lazy, I refer to the code in Listing 3-2 as a *method.* If I'm not being lazy, I refer to this code as a *method declaration.*

- The method declaration in Listing 3-2 has two parts. The first line (the part with `fixTheAlternator` in it, up to but not including the open curly brace) is a *method header*. The rest of Listing 3-2 (the part surrounded by curly braces) is a *method body*.
- The term *method declaration* distinguishes the list of instructions in Listing 3-2 from the instruction in Listing 3-3, which is known as a *method call*.

REMEMBER

A *method's declaration* tells the computer what happens if you call the method into action. A *method call* (a separate piece of code) tells the computer to call the method into action. A method's declaration and the method's call tend to be in different parts of the Java program.

The main method in a program

Figure 3-2 has a copy of the code from Listing 3-1. The code contains the declaration of a method named `main`. (Just look for the word *main* in the code's method header.) For now, don't worry about the other word — `void` — and the empty parentheses in the method header. I explain these features in Chapter 7.

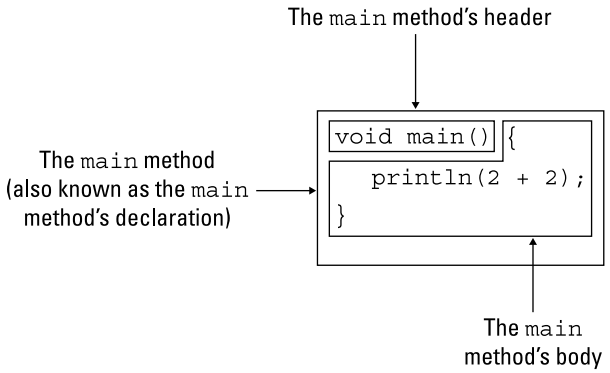

The main method's header
The main method (also known as the main method's declaration)
void main() {
println(2 + 2);
}
The main method's body

FIGURE 3-2: The `main` method.

Like any Java method, the `main` method is a recipe:

```
How to make biscuits:
    Heat the oven.
    Roll the dough.
    Bake the rolled dough.
```

or

```
How to follow the main instruction(s) in Figure 3–2:
    Print the numerical value of 2 + 2 on the screen.
```

The word *main* plays a special role in Java. In particular, you never write code that explicitly calls a `main` method into action. The word *main* is the name of the method that is called into action automatically when the program begins running.

Look back at Figure 3-1. When the Two Plus Two program runs, the computer automatically finds the program's `main` method and executes any instructions inside the method's body. The `main` method's body has only one instruction. That instruction tells the computer to print the value of `2 + 2` on the screen. So, in Figure 3-1, `4` appears on the computer screen.

REMEMBER

The instructions in a method aren't executed until the method is called into action. But, if you give a method the name *main*, that method is called into action automatically.

TECHNICAL STUFF

Almost every computer programming language has something akin to Java's methods. If you've worked with other languages, you may remember terms like *functions*, *procedures*, *subprograms*, *subroutines*, and good ol' `PERFORM` statements. Whatever you call it in your favorite programming language, a method is a bunch of instructions collected and given a new name.

How you finally tell the computer to do something

The heart of Listing 3-1 is the single line that issues a direct instruction to the computer. The line, which is highlighted in Figure 3-3, tells the computer to display the value of `2 + 2`. This line is a statement. In Java, a *statement* is a direct instruction that tells the computer to do something (for example, display this text, put a certain number in that memory location, make a window appear, or whatever).

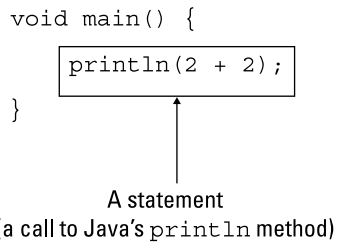


FIGURE 3-3: A Java statement.

Of course, Java has different kinds of statements. A method call, which I introduce in the earlier section "The Java method," is one of the many kinds of Java

statements. Listing 3-3 shows you what a method call looks like, and Figure 3-3 also contains a method call that looks like this:

```
println(2 + 2);
```

When the computer executes this statement, the computer calls into action a method named `println`.

WARNING

I said it already, but it's worth repeating: In `println`, the next-to-last character is a lowercase letter *l* (as in the word *line*), not a digit *1* (as in the number *one*). If you use a digit *1*, your code won't work. Just think of `println` as a way of saying "print line" and you won't have any problem.

Figure 3-4 illustrates the `println` situation. Actually, two methods play active roles in the running of this chapter's Two Plus Two program. Here's how they work:

- **There's a declaration for a `main` method.** I wrote the `main` method myself. This `main` method is called automatically whenever I run the program.
- **There's a call to the `println` method.** The method call for the `println` method is the only statement in the body of the `main` method. In other words, calling the `println` method is the only item on the `main` method's to-do list.

  The declaration for the `println` method is buried inside the official Java API. For a refresher on the Java API, see the sections "The grammar and the common names" and "The words in a Java program," earlier in this chapter.

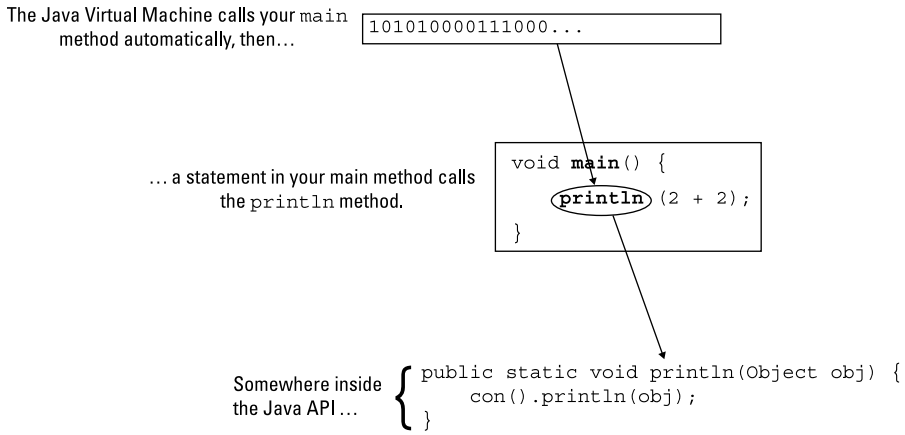

The Java Virtual Machine calls your main method automatically, then…
101010000111000...
… a statement in your main method calls the println method.
void main() {
println (2 + 2);
}
Somewhere inside the Java API …
public static void println(Object obj) {
con().println(obj);
}

FIGURE 3-4: Calling the `println` method.

In Figure 3-4, each pair of parentheses surround a *parameter list*. With parameters, a method call communicates on-the-fly information to its method declaration. For example, in Figure 3-4, the method call says, "print what you get when you add 2 plus 2." In a different program, a method call might say, "print the words *Hello World*" or "print the complete works of Shakespeare."

In Figure 3-4, notice how the `main` and `con` method calls have nothing in their parameter lists. Sometimes, when you call a method, you have no extra information to send. That's okay.

You can read more about parameter lists in Chapter 7.

TECHNICAL STUFF

When I say things like, "`println` is buried inside the API," I'm not doing justice to the API. True, you can ignore all the nitty-gritty Java code inside the API. All you need to remember is that `println` is defined somewhere inside that code. But I'm not being fair when I make the API code sound like something magical. The API is just another bunch of Java code. The statements in the API that tell the computer what it means to carry out a call to `println` look a lot like the Java code in Listing 3-1.

In Java, each statement (like the boxed line in Figure 3-3) ends with a semicolon. The other lines in Figure 3-3 don't end with semicolons, because the other lines in Figure 3-3 aren't statements. For instance, the method header (the line with the word *main* in it) doesn't directly tell the computer to do anything. The method header announces, "Just in case you ever want to do `main`, the next few lines of code tell you how to do it."

REMEMBER

Every complete Java statement ends with a semicolon.

Brace yourself

Long ago, or maybe not so long ago, your schoolteachers told you how useful outlines are. With an outline, you can organize thoughts and ideas, help people see forests instead of trees, and generally show that you're a member of the Tidy Persons Club. Well, a Java program is like an outline. The program in Listing 3-1 starts with a header line that says, "Here comes a method named `main`." After that header, a subheader announces, "Here comes a call to the `println` method."

Now, if a Java program is like an outline, why doesn't a program *look* like an outline? What takes the place of the Roman numerals, capital letters, and other items? The answer is twofold:

- In a Java program, curly braces enclose meaningful units of code.
- You, the programmer, can (and should) indent lines so that other programmers can see at a glance the outline form of your code.

In an outline, everything is subordinate to the item in Roman numeral *I.* In a Java program, everything is subordinate to the top line. In Listing 3-1, the top line is the line with the word *main* in it. To indicate that everything else in the code is subordinate to this `main` line, you use curly braces. Everything else in the code goes inside these curly braces.

In Chapter 7, when you start using classes, you put a program's `main` method inside its own class. Look at Listing 3-4. In that listing, one pair of curly braces indicates that the `main` method is subordinate to a class named `TwoPlusTwo`. Another pair of curly braces indicates that the call to `println` is, in turn, subordinate to the `main` method's header line.

LISTING 3-4: **Curly Braces for a Java Class**

```
import static io.IO.println; // Ignore this line for now
                             // (or skip to Chapter 11).
class TwoPlusTwo {
    void main() {
        println(2 + 2);
    }
}
```

In an outline, some items are at the bottom of the food chain. In Listing 3-4, the corresponding line is the line that begins with `println`. Accordingly, this `println` line goes inside all the other curly braces and is indented more than any other line.

REMEMBER

Never lose sight of the fact that a Java program is, first and foremost, an outline.

If you put curly braces in the wrong places or omit curly braces where the braces should be, your program probably won't work. If your program works, it'll probably work incorrectly.

If you don't indent lines of code in an informative manner, your program will still work correctly, but neither you nor any other programmer will be able to figure out what you were thinking when you wrote the code.

If you're a visual thinker, you can picture outlines of Java programs in your head. One friend of mine visualizes an actual numbered outline morphing into a Java program. (See Figure 3-5.) Another person, who shall remain nameless, uses more bizarre imagery. (See Figure 3-6.)

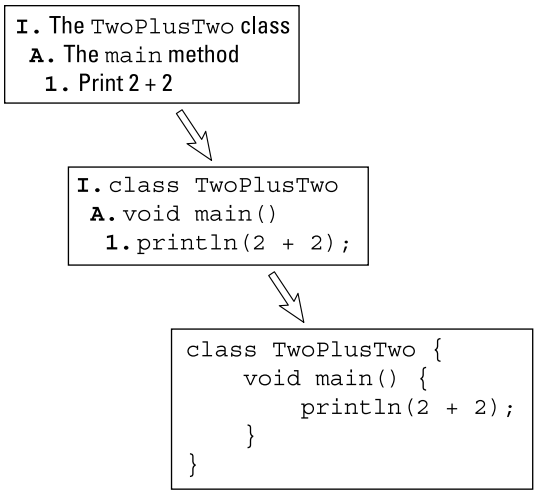


FIGURE 3-5: An outline turns into a Java program.

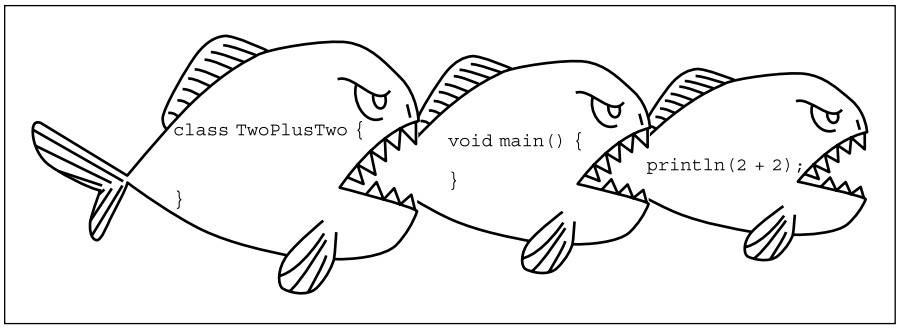


FIGURE 3-6: A class is bigger than a method; a method is bigger than a statement.

TIP

I appreciate a good excuse as much as the next guy, but failing to indent your Java code is inexcusable. In fact, many IDEs have tools to indent your code automatically. In your favorite IDE, look for menus such as Source➪Format or Code➪Reformat Code.

TRY IT OUT

Here are some things for you to try to help you understand the material in this section. If trying these things builds your confidence, that's good. If trying these things makes you question what you've read, that's good too. If trying these things makes you nervous, don't be discouraged. You can find answers and other help at this book's website (`https://JavaForDummies.allmycode.com`). You can also email me with your questions (`JavaForDummies@allmycode.com`).

BEGIN THE BEGUINE

» If you've downloaded the code from this book's website, import Listing 3-1 (from the downloaded `03–01` folder) into your IDE. If you don't plan to

download the code, create a new project in your IDE. In the new project, create a Java file containing the code from Listing 3-1. With the downloaded code, or with your own, newly-created code, run the program and look for the number `4` in the output.

- Try running the code in Listing 3-1 with the text `2 + 2` changed to "Four". (Remember to surround `Four` with quotation marks (").) What happens?
- Try running the code in Listing 3-1 with the text `2 + 2` changed to Four. (In this example, try *not* surrounding `Four` with quotation marks (").) What happens?

CAPITAL PAINS

- Try to run the code in Listing 3-1 with the word `main` (all lowercase) changed to `Main` (starting with an uppercase letter). What happens?
- Try to run the code in Listing 3-1 with the word `println` (all lowercase) changed to `PrintLn` (with two uppercase letters). What happens?
- In Listing 3-1, change the line

```
println(2 + 2);
```

to

```
System.out.println(2 + 2);
```

Make sure that this modified code runs.

- Following up on the previous experiment, try to run the code in Listing 3-1 with the word `System` (starting with an uppercase letter) changed to `system` (all lowercase). What happens?

STOLEN SEMICOLON

- Try to run the code in Listing 3-1 with the semicolon missing. What happens?
- Try to run the code in Listing 3-1 with additional semicolons added at the ends of some of the lines. What happens?

MISCELLANEOUS MISCHIEF

- Try to run the code in Listing 3-1 with the indentation changed. For example, don't indent the middle line. What happens?
- Try to run the code in Listing 3-1 with no line breaks. Put all the code on one long line. What happens?

» Try to run the code in Listing 3-1 with the word `println` changed to `print1n` (with the digit 1 near the end). What happens?

» Try to run the code in Listing 3-1 with the text `2 + 2` changed to `"Use a straight quote \", not a curly quote \u201D"`. (Remember to type all three quotation marks (").) What happens?

And Now, a Few Comments

People gather around campfires to hear the old legend about a programmer whose laziness got her into trouble. To maintain this programmer's anonymity, I call her Jane Pro. Jane worked many months to create the holy grail of computing: a program that thinks on its own. If completed, this program could work independently, learning new concepts without human intervention. Day after day, night after night, Jane Pro labored to give the program that spark of creative, independent thought.

One day, when Jane Pro was almost finished with the project, she received a disturbing piece of paper mail from her health insurance company. No, the mail wasn't about a serious illness. It was about a routine office visit. The insurance company's claim form had a place for Jane's date of birth, as though her date of birth had changed since the last time she sent in a claim. She had absentmindedly scribbled *2016* as her year of birth, so the insurance company refused to pay the bill.

Jane dialed the insurance company's phone number. Within 20 minutes, she was talking to a live person. "I'm sorry," said the live person. "To resolve this issue, you must dial a different number." Well, you can guess what happened next. "I'm sorry. The other operator gave you the wrong number." And then, "I'm sorry. You must call back the original phone number."

Five months later, Jane's ear ached, but after 800 hours on the phone, she had finally gotten a tentative promise that the insurance company would eventually reprocess the claim. Elated as she was, she was anxious to get back to her programming project. Could she remember what all those lines of code were supposed to be doing?

No, she couldn't. Jane stared and stared at her own work and, like a dream that doesn't make sense the next morning, the code was completely meaningless to her. She had written a million lines of code, and not one line was accompanied by an informative explanatory comment. She had left no clues to help her understand what she'd been thinking, so in frustration she abandoned the whole project.

Adding comments to your code

Listing 3-5 holds an enhanced version of this chapter's sample program. In addition to all the keywords, identifiers, and punctuation, Listing 3-5 has text that's meant for human beings to read.

LISTING 3-5: Three Kinds of Comments

```
/*
 * Listing 3-5 in "Java For Dummies, 9th Edition"
 *
 * Copyright 2024 Wiley Publishing, Inc.
 * All rights reserved.
 */


/**
 * This code adds 2 and 2.
 *
 * @author Barry Burd
 * @version 1.0 6/24/24
 * @see java.io.IO
 */


/**
 * The main method is where execution of the code begins.
 * This main method uses Java 23 preview features. These
 * features might not be available in future Java releases.
 */


void main() {
    println(2 + 2);  //Replace 2 with Math.sqrt(2)
}
```

A *comment* is a special section of text, inside a program, whose purpose is to help people understand the program. A comment is part of a good program's documentation.

The Java programming language has three kinds of comments:

- **Traditional comments:** The first six lines of Listing 3-5 form one *traditional* comment. The comment begins with `/*` and ends with `*/`. Everything between the opening `/*` and the closing `*/` is for human eyes only. No

CROSS REFERENCE

information about `"Java For Dummies, 9th Edition"` or `Wiley Publishing, Inc.` is translated by the compiler.

To read about compilers, see Chapter 2.

The second, third, fourth, and fifth lines in Listing 3-5 have extra asterisks (*). I call them "extra" because these asterisks aren't required when you create a comment. They just make the comment look pretty. I include them in Listing 3-5 because, for some reason that I don't entirely understand, most Java programmers add these extra asterisks.

- **End-of-line comments:** The text `//Replace 2 with Math.sqrt(2)` in Listing 3-5 is an end-of-line comment. An *end-of-line* comment starts with two slashes and goes to the end of a line of type. Once again, the compiler doesn't translate the text inside the end-of-line comment.
- **Javadoc comments:** A *Javadoc* comment begins with a slash and two asterisks (`/**`). Listing 3-5 has two Javadoc comments: One starts with the text `This code adds 2 and 2.` and another starts with `The main method is where...`.

  A *Javadoc* comment, which is a special kind of traditional comment, is meant to be read by people who never even look at the Java code. But that doesn't make sense. How can you see the Javadoc comments in Listing 3-5 if you never look at Listing 3-5?

  Well, a certain program called *Javadoc* (what else?) can find all the Javadoc comments in Listing 3-5 and turn these comments into a nice-looking web page. Figure 3-7 shows the page.

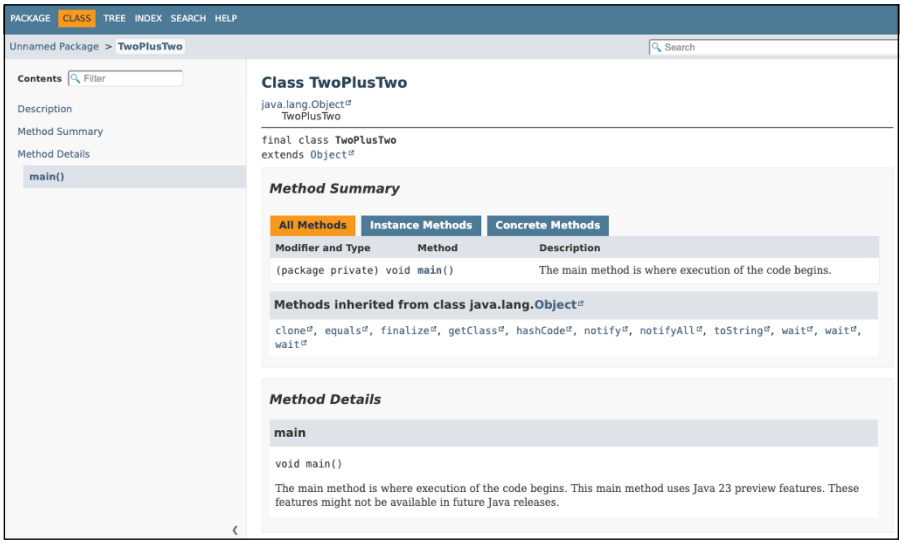


FIGURE 3-7: The Javadoc page generated from the code in Listing 3-5.

Javadoc comments are great. Here are several great things about them:

- The only person who has to look at a piece of Java code is the programmer who writes the code. Other people who use the code can find out what the code does by viewing the automatically generated web page.
- Because other people don't look at the Java code, other people don't make changes to the Java code. (In other words, other people don't introduce errors into the existing Java code.)
- Because other people don't look at the Java code, other people don't have to decipher the inner workings of the Java code. All these people need to know about the code is what they read on the code's web page.
- The programmer doesn't create two separate files — some Java code over here and some documentation about the code over there. Instead, the programmer creates one piece of Java code and embeds the documentation (in the form of Javadoc comments) right inside the code.
- The generation of web pages from Javadoc comments is automatic. So everyone's documentation has the same format. No matter whose Java code you use, you find out about that code by reading a page like the one in Figure 3-7. That's good because the format in Figure 3-7 is familiar to anyone who uses Java.
- Best of all, Java's online API Specification, which describes all of Java's standard identifiers, comes entirely from runs of the *Javadoc* program on comments in Java code. There's nothing arbitrary or haphazard about Java's API Specification. The Specification is always in sync with the actual Java code.

TIP

You can generate your own web pages from the Javadoc comments that you put in your code. From your IDE's menu bar, choose Tools or Project and look for a command labeled Generate JavaDoc.

What's Barry's excuse?

For years I've been telling my students to put comments in their code, and for years I've been creating sample code (like the code in Listing 3-1) with no comments in it. Why?

Three little words: *Know your audience.* When you write complicated, real-life code, your audience is other programmers, information technology managers, and people who need help deciphering what you've done. When I write simple samples of code for this book, my audience is you — the novice Java programmer.

Rather than read my comments, your best strategy is to stare at my Java statements — the statements that Java's compiler deciphers. That's why I put so few comments in this book's listings.

Besides, I'm a little lazy.

Using comments to experiment with your code

You may hear programmers talk about *commenting out* certain parts of their code. When you're writing a program and something's not working correctly, it often helps to try removing some of the code. If nothing else, you find out what happens when that suspicious code is removed. Of course, you may not like what happens when the code is removed, so you don't want to delete the code completely. Instead, you turn your ordinary Java statements into comments. For instance, you turn the statement

```
println(2 + 2);
```

into the comment

```
//println(2 + 2);
```

This change keeps the Java compiler from seeing the code while you try to figure out what's wrong with your program.

Traditional comments can be tricky when you're trying to comment out code. The big problem is that you can't put one traditional comment inside another. Suppose that you want to comment out the following statements:

```
println(2 + 2);
println(2 + 3);
 /*
  * Intentionally displays on four separate lines
  */
println(2 + 4);
println(2 + 5);
```

If you try to turn this code into one traditional comment, you get the following mess:

```
/*
  println(2 + 2);
  println(2 + 3);
  /*
   * Intentionally displays on four separate lines
   */
  println(2 + 4);
  println(2 + 5);
*/
```

The first `*/` (after `Intentionally displays`) ends the traditional comment prematurely. Then the `2 + 4` and `2 + 5` statements aren't commented out, and the last `*/` chokes the compiler.

TIP

Most IDEs can comment out sections of your code for you automatically. From your IDE's menu bar, choose Code or Source and look for menu items pertaining to comments.

2 Writing Your Own Java Programs

IN THIS PART . . .

Create new values and modify existing values.

Add decision-making to your application's logic.

Use repetition as a tool in problem-solving.

IN THIS CHAPTER

- Assigning values to things
- Making things store certain types of values
- Applying operators to get new values

Chapter 4

Making the Most of Variables and Their Values

The following conversation between Mr. Van Doren and Mr. Barasch never took place:

Charles: A sea squirt eats its brain, turning itself from an animal into a plant.

Jack: Is that your final answer, Charles?

Charles: Yes, it is.

Jack: How much money do you have in your account today, Charles?

Charles: I have fifty dollars and twenty-two cents in my checking account.

Jack: Well, you had better call the IRS, because your sea squirt answer is correct. You just won a million dollars to add to your checking account. What do you think of that, Charles?

Charles: I owe it all to honesty, diligence, and hard work, Jack.

Some aspects of this dialogue can be represented in Java by a few lines of code.

Varying a Variable

No matter how you acquire your million dollars, you can use a variable to tally your wealth. Listing 4-1 shows the code.

LISTING 4-1: **Using a Variable**

```
amountInAccount = 50.22;
amountInAccount = amountInAccount + 1000000.00;
```

WARNING

The code in Listing 4-1 isn't a complete Java program. If you try to run this code the same way you run the example in Chapter 3, you'll get an error message. For a complete, runnable program, see Listing 4-2.

The code in Listing 4-1 makes use of the `amountInAccount` variable. A *variable* is a placeholder. You can stick a number like 50.22 into a variable. After you place a number in the variable, you can change your mind and put a different number into the variable. (That's what varies in a variable.) Of course, when you put a new number in a variable, the old number is no longer there. If you didn't save the old number somewhere else, the old number is gone.

Figure 4-1 gives a before-and-after picture of the code in Listing 4-1. After the first statement in Listing 4-1 is executed, the variable `amountInAccount` has the number 50.22 in it. Then, after the second statement of Listing 4-1 is executed, the `amountInAccount` variable suddenly has 1000050.22 in it. When you think about a variable, picture a place in the computer's memory where wires and transistors store 50.22, 1000050.22, or whatever. On the left side of Figure 4-1, imagine that the box with 50.22 in it is surrounded by millions of other such boxes.

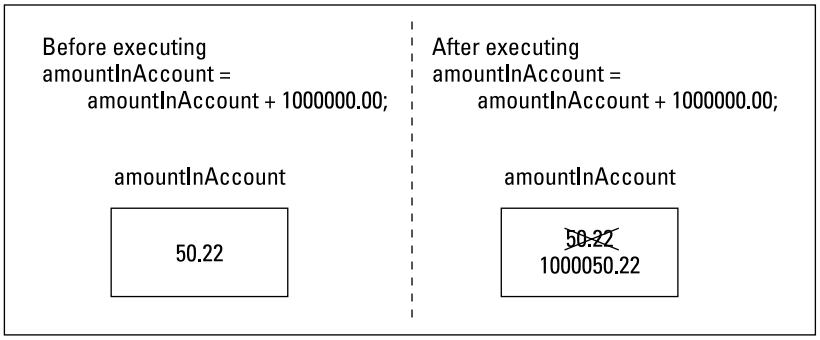


FIGURE 4-1: A variable (before and after).

Now you need some terminology. The thing stored in a variable is a *value.* A variable's value can change during the run of a program (when Jack gives you a million bucks, for instance). The value that's stored in a variable isn't necessarily a number. (For instance, you can create a variable that always stores a letter.) The kind of value that's stored in a variable is a variable's *type.*

You can read more about types in the section "The types of values that variables may have," later in this chapter.

TECHNICAL STUFF

A subtle, almost unnoticeable difference exists between a variable and a variable's *name.* Even in formal writing, I often use the word *variable* when I mean *variable name.* Strictly speaking, `amountInAccount` is a variable name, and all the memory storage associated with `amountInAccount` (including the type that `amountInAccount` has and whatever value `amountInAccount` currently represents) is the variable itself. If you think this distinction between *variable* and *variable name* is too subtle for you to worry about, join the club.

Every variable name is an identifier — a name that you can make up in your own code. In preparing Listing 4-1, I made up the name *amountInAccount.*

CROSS REFERENCE

For more information on the kinds of names in a Java program, see Chapter 3.

Before the sun sets on Listing 4-1, you need to notice one more part of the listing. The listing has `50.22` and `1000000.00` in it. Anybody in their right mind would call these things *numbers,* but in a Java program it helps to call these things *literals.*

And what's so literal about `50.22` and `1000000.00`? Well, think about the variable `amountInAccount` in Listing 4-1. The variable `amountInAccount` stands for 50.22 some of the time, but it stands for 1000050.22 the rest of the time. You could use the word *number* to talk about `amountInAccount`. But really, what `amountInAccount` stands for depends on the fashion of the moment. On the other hand, 50.22 literally stands for the value $50\frac{22}{100}$.

TIP

You can add underscores to numeric literals. Rather than use the plain old `1000000.00` in Listing 4-1, you can write `amountInAccount = amountInAccount + 1_000_000.00`. Unfortunately, you can't easily do what you're most tempted to do. You can't write `1,000,000.00` (as you would in the United States), nor can you write `1.000.000,00` (as you would in Germany). If you want to display a number such as `1,000,000.00` in the program's output, you have to use some fancy formatting tricks. For more information about formatting, check Chapters 7 and 12.

Assignment statements

Statements like the ones in Listing 4-1 are called assignment statements. In an *assignment statement*, you assign a value to something. In many cases, this something is a variable.

I recommend getting into the habit of reading assignment statements from right to left. Figure 4-2 illustrates the action of the first line in Listing 4-1.

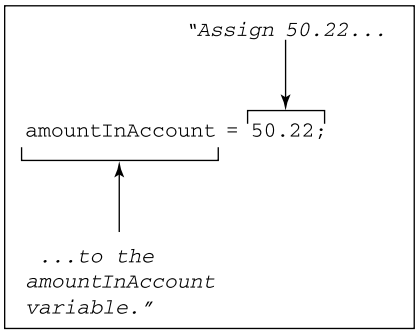


FIGURE 4-2: The action of the first line in Listing 4-1.

The second line in Listing 4-1 is just a bit more complicated. Figure 4-3 illustrates the action of the second line in Listing 4-1.

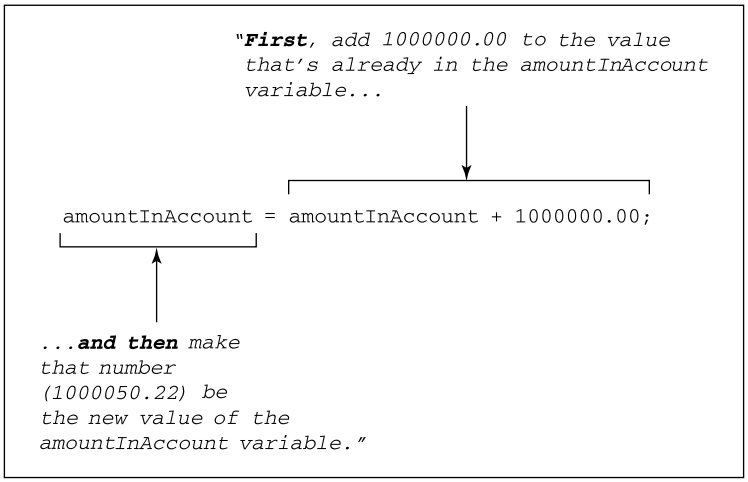

"First, add 1000000.00 to the value that's already in the amountInAccount variable...
amountInAccount = amountInAccount + 1000000.00;
...and then make that number (1000050.22) be the new value of the amountInAccount variable."

FIGURE 4-3: The action of the second line in Listing 4-1.

REMEMBER

In an assignment statement, the thing being assigned a value is always on the left side of the equal sign.

The types of values that variables may have

Have you seen the TV commercials that make you think you're flying among the circuits inside a computer? Pretty cool, eh? These commercials show 0s (zeros) and 1s (ones) sailing by because 0s and 1s are the only things that computers can deal with. When you think a computer is storing the letter *J*, the computer is really storing 01001010. Everything inside the computer is a sequence of 0s and 1s. As every computer geek knows, a 0 or 1 is called a *bit.*

As it turns out, the sequence 01001010, which stands for the letter *J*, can also stand for the number 74. The same sequence can also stand for $1.0369608636003646 \times 10^{-43}$. In fact, if the bits are interpreted as screen pixels, the same sequence can be used to represent the dots shown in Figure 4-4. The meaning of 01001010 depends on the way the software interprets this sequence of 0s and 1s.

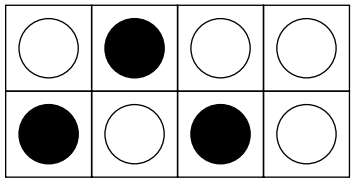


FIGURE 4-4: An extreme close-up of eight black-and-white screen pixels.

How do you tell the computer what 01001010 stands for? The answer is in the concept of type. The *type* of a variable is the range of values that the variable is permitted to store. I copied the lines from Listing 4-1 and put them into a complete Java program. The program is in Listing 4-2. When I run the program in Listing 4-2, I get the output shown in Figure 4-5.

```
You have $1000050.22 in your account.
```

FIGURE 4-5: Running the program in Listing 4-2.

LISTING 4-2: **Adding a Million to Your Account**

```
void main() {
    double amountInAccount;

    amountInAccount = 50.22;
    amountInAccount = amountInAccount + 1000000.00;

    print("You have $");
    print(amountInAccount);
    println(" in your account.");
}
```

In Listing 4-2, look at the first line in the body of the main method:

```
double amountInAccount;
```

This line is called a *variable declaration.* Putting this line in your program is like saying, "I'm declaring my intention to have a variable named *amountInAccount* in my program." This line reserves the name *amountInAccount* for your use in the program.

In this variable declaration, the word *double* is a Java keyword. This word *double* tells the computer what kinds of values you intend to store in `amountInAccount`. In particular, the word *double* stands for numbers between -1.8×10^{308} and 1.8×10^{308}. (These are enormous numbers with 308 zeros before the decimal point. Only the world's richest people write checks with 308 zeros in them. The second of these numbers is one-point-eight gazazzo-zillion-kaskillion. The number 1.8×10^{308}, a constant defined by the International Bureau of Weights and Measures, is the number of eccentric computer programmers between Sunnyvale, California, and the M31 Andromeda Galaxy.)

More important than the humongous range of the `double` keyword's numbers is the fact that a `double` value can have digits beyond the decimal point. After you declare `amountInAccount` to be of type `double`, you can store all sorts of numbers in `amountInAccount`. You can store `50.22`, `0.02398479`, or `-3.0`. In Listing 4-2, if I hadn't declared `amountInAccount` to be of type `double`, I may not have been able to store `50.22`. Instead, I would have had to store plain old `50`, with no digits beyond the decimal point.

Another type — type `float` — also allows you to have digits beyond the decimal point. But `float` values aren't as accurate as `double` values.

TIP

In many situations, you have a choice. You can declare certain values to be either `float` values or `double` values. But don't sweat the choice between `float` and `double`. For most programs, just use `double`. With today's fancy processors, the space you save using the `float` type is almost never worth the loss of accuracy. (For more details, see the nearby sidebar, "To the decimal point and beyond!")

The big million-dollar jackpot in Listing 4-2 is impressive. But Listing 4-2 doesn't illustrate the best way to deal with dollar amounts. In a Java program, the best way to represent currency is to shun the `double` and `float` types and opt instead for a type named `BigDecimal`. For more information, see this book's website (`https://JavaForDummies.allmycode.com`).

TO THE DECIMAL POINT AND BEYOND!

Java has two different types that have digits beyond the decimal point: type `double` and type `float`. So, what's the difference? When you declare a variable to be of type `double`, you're telling the computer to keep track of 64 bits when it stores the variable's values. When you declare a variable to be of type `float`, the computer keeps track of only 32 bits.

You could change Listing 4-2 and declare `amountInAccount` to be of type `float`:

```
float amountInAccount;
```

Surely, 32 bits are enough to store a small number like 50.22, right? Well, they are and they aren't. You could easily store 50.00 with only 32 bits. Heck, you could store 50.00 with only 6 bits. The size of the number doesn't matter. The accuracy matters. In a 64-bit double variable, you're using most of the bits to store stuff beyond the decimal point. To store the .22 part of 50.22, you need more than the measly 32 bits that you get with type `float`.

Do you really believe what you just read — that it takes more than 32 bits to store .22? To help convince you, I made a few changes to the code in Listing 4-2. I made `amountInAccount` be of type `float`. Then I changed the first three statements inside the main method:

```
float amountInAccount;
amountInAccount = 50.22F;
amountInAccount = amountInAccount + 1000000.00F;
```

(To understand why I used the letter `F` in `50.22F` and `1000000.00F`, see Table 4-1, later in this chapter.) The output I got was

```
You have $1000050.25 in your account.
```

Compare this with the output in Figure 4-5. When I switch from type `double` to type `float`, Charles has an extra three cents in his account. By changing to the 32-bit `float` type, I've clobbered the accuracy in the `amountInAccount` variable's hundredths place. That's bad.

Another difficulty with `float` values is purely cosmetic. Look again at the literals, `50.22` and `1000000.00`, in Listing 4-2. The Laws of Java say that literals like these take up 64 bits each. So, if you declare `amountInAccount` to be of type `float`, you'll run into

(continued)

(continued)

trouble. You'll have trouble stuffing those 64-bit literals into your little 32-bit `amountInAccount` variable. To compensate, you can switch from `double` literals to `float` literals by adding an `F` to each `double` literal, but a number with an extra `F` at the end looks funny.

```
float amountInAccount;
amountInAccount = 50.22F;
amountInAccount = amountInAccount + 1000000.00F;
```

To experiment with numbers, visit `https://www.binaryconvert.com/`. The page takes any number you enter and shows you how the number would be represented as 32 bits and as 64 bits.

How to hold the line

The last three statements in Listing 4-2 use a neat formatting trick. You want to display several different items on a single line on the screen. You put these items in separate statements. All but the last of the statements are calls to `print`. (The last statement is a call to `println`.) Calls to `print` display text on part of a line and then leave the cursor at the end of the current line. After executing `print`, the cursor is still at the end of the same line, so the next `print` or `println` can continue printing on that same line. With several calls to `print` capped off by a single call to `println`, the result is just one nice-looking line of output. (Refer to Figure 4-5.)

REMEMBER

A call to `print` writes some things and leaves the cursor sitting at the end of the line of output. A call to `println` writes things and then finishes the job by moving the cursor to the start of a brand-new line of output.

TRY IT OUT

Run the code in Listing 4-2 to make sure that it runs correctly on your computer. Then see what happens when you make the following changes:

NUMBER FORMAT EXPERIMENTS

- Add thousands-separators to the number `1000000.00` in the code. For example, if you live in the United States, where the thousands-separator is a comma, change the number to `1,000,000.00` and see what happens. (Hint: Nothing good happens.)
- Try using underscores as thousands-separators in the code. That is, change `1000000.00` to `1_000_000.00` and see what happens.

» Add a currency symbol to the number `50.22` in the code. For example, if you live in the United States, where the currency symbol is $, see what happens when you change the first assignment statement to `amountInAccount = $50.22`.

HOW TO DISPLAY VALUES

» Listing 4-2 has two `print` statements and one `println` statement. Change all three to `println` statements and then run the program.

» The code in Listing 4-2 displays one line of text in its output. Using the `amountInAccount` variable, add statements to the program so that it displays a second line of text. Have the second line of text be "Now you have even more! You have 2000000.00 in your account."

Numbers without decimal points

"In 1995, the average family had 2.3 children."

At this point, a wise guy always remarks that no real family has exactly 2.3 children. Clearly, whole numbers have a role in this world. Therefore, in Java, you can declare a variable to store nothing but whole numbers. Listing 4-3 shows a program that uses whole number variables.

LISTING 4-3: **Stuffing People in an Elevator**

```
void main() {
    int weightOfAPerson;
    int elevatorWeightLimit;
    int numberOfPeople;

    weightOfAPerson = 150;
    elevatorWeightLimit = 1400;
    numberOfPeople = elevatorWeightLimit / weightOfAPerson;

    print("You can fit ");
    print(numberOfPeople);
    println(" people on the elevator.");
}
```

The story behind the program in Listing 4-3 takes some heavy-duty explaining. Here goes:

You have a hotel elevator whose weight capacity is 1,400 pounds. One weekend the hotel hosts the Brickenchicker family reunion. A certain branch of the Brickenchicker family has been blessed with identical decuplets (ten siblings, all with the same physical characteristics). Normally, each of the Brickenchicker decuplets weighs exactly 145 pounds. But on Saturday the family has a big catered lunch, and, because lunch included strawberry shortcake, each of the Brickenchicker decuplets now weighs 150 pounds. Immediately after lunch, all ten of the Brickenchicker decuplets arrive at the elevator at exactly the same time. (Why not? All ten of them think alike.) So, the question is, how many of the decuplets can fit on the elevator?

Now remember, if you put one ounce more than 1,400 pounds of weight on the elevator, the elevator cable breaks, plunging all decuplets on the elevator to their sudden (and costly) deaths.

The answer to the Brickenchicker riddle (the output of the program of Listing 4-3) is shown in Figure 4-6.

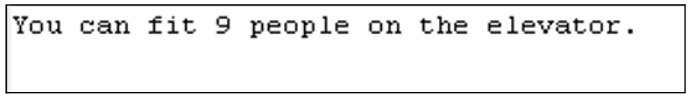


FIGURE 4-6: Save the Brickenchickers.

At the core of the Brickenchicker elevator problem, you have whole numbers — numbers with no digits beyond the decimal point. When you divide 1,400 by 150, you get 9⅓, but you shouldn't take the ⅓ seriously. No matter how hard you try, you can't squeeze an extra 50 pounds' worth of Brickenchicker decuplet onto the elevator. This fact is reflected nicely in Java. In Listing 4-3, all three variables (`weightOfAPerson`, `elevatorWeightLimit`, and `numberOfPeople`) are of type `int`. An `int` value is a whole number. When you divide one `int` value by another (as you do with the slash in Listing 4-3), you get another `int`. When you divide 1,400 by 150, you get 9 — not 9⅓. You see this in Figure 4-6. Taken together, the following statements display 9 onscreen:

```
numberOfPeople = elevatorWeightLimit / weightOfAPerson;

print(numberOfPeople);
```

FOUR WAYS TO STORE WHOLE NUMBERS

Java has four types of whole numbers. The types are `byte`, `short`, `int`, and `long`. Unlike the complicated story about the accuracy of types `float` and `double`, the only thing that matters when you choose among the whole number types is the size of the number you're trying to store. If you want to use numbers larger than 127, don't use `byte`. To store numbers larger than 32767, don't use `short`.

Most of the time, you'll use `int`. But if you need to store numbers larger than 2147483647, forsake `int` in favor of `long`. (A `long` number can be as big as 9223372036854775807.) For the whole story, see Table 4-1, a little later in this chapter.

TAKE A FLYING LEAP

TRY IT OUT

My wife and I were married on February 29, so we have one anniversary every four years. Write a program with a variable named `years`. Based on the value of the `years` variable, the program displays the number of anniversaries we've had. For example, if the value of `years` is 4, the program displays the sentence `Number of anniversaries: 1`. If the value of `years` is 7, the program still displays `Number of anniversaries: 1`. But if the value of years is 8, the program displays `Number of anniversaries: 2`.

Combining declarations and initializing variables

Look back at Listing 4-3. In that listing, you see three variable declarations — one for each of the program's three `int` variables. I could have done the same thing with just one declaration:

```
int weightOfAPerson, elevatorWeightLimit, numberOfPeople;
```

WARNING

If two variables have completely different types, you can't create both variables in the same declaration. For instance, to create an `int` variable named *weightOfFred* and a `double` variable named *amountInFredsAccount*, you need two separate variable declarations.

You can give variables their starting values in a declaration. In Listing 4-3, for instance, one declaration can replace several lines in the `main` method (all but the calls to `print` and `println`):

```
int weightOfAPerson = 150, elevatorWeightLimit = 1400,
    numberOfPeople = elevatorWeightLimit / weightOfAPerson;
```

When you do this, you don't say that you're assigning values to variables. The pieces of the declarations with equal signs in them aren't really called assignment statements. Instead, you say that you're *initializing* the variables. Believe it or not, keeping this distinction in mind is helpful.

For example, when you initialize a variable inside a program's `main` method, you don't have to specify the new variable's type. You can start Listing 4-2 like this:

```
void main() {
    var amountInAccount = 0.0;
```

If you want, you can start Listing 4-3 this way:

```
void main() {
    var weightOfAPerson = 150;
    var elevatorWeightLimit = 1400;
    var numberOfPeople = elevatorWeightLimit / weightOfAPerson;
```

In either case, the word `var` replaces the name of a type. This trick works because Java is smart. When you write

```
var amountInAccount = 0.0;
```

Java looks at the number `0.0` and realizes that `amountInAccount` is a `double` value. It's a number with digits to the right of the decimal point. In the same way, Java sees

```
var weightOfAPerson = 150;
```

and figures out on its own that `weightOfAPerson` has to be an `int` value.

This `var` business doesn't work if you don't initialize your new variable. For example, the lines

```
// BAD CODE:
var numberOfCats;
numberOfCats = 3;
```

are unacceptable as far as Java is concerned. Java can't wait until the assignment statement `numberOfCats = 3` to decide what type of variable `numberOfCats` is. Java wants to know immediately.

WARNING

There are other situations in which the use of `var` is illegal, even with an initialization. For details, see Chapter 7.

Like everything else in life, initializing a variable has advantages and disadvantages:

- **When you combine six lines of Listing 4-3 into just one declaration, the code becomes more concise.** Sometimes concise code is easier to read. Sometimes it's not. As a programmer, it's your judgment call.
- **By initializing a variable, you might automatically avoid certain programming errors.** For an example, see Chapter 7.
- **In some situations, you have no choice. The nature of your code forces you either to initialize or not to initialize.** For an example that doesn't lend itself to variable initialization, see the deleting-evidence program in Chapter 6.

Experimenting with JShell

Listing 4-1 comes with a big warning. The listing is missing its `void main()` header, so the code in the listing is incomplete. And yet, the text in Listing 4-1 isn't entirely meaningless. You should be able to experiment with these two lines of code.

And, in fact, you can! Java comes with a tool named *JShell*. With one tiny modification, you can use JShell to run the bare-bones code in Listing 4-1. In fact, JShell allows you to experiment interactively with all kinds of code snippets. When you use JShell, you hardly ever type an entire program. Instead, you type a Java statement, and then JShell responds to your statement, and then you type a second statement, and then JShell responds to your second statement, and then you type a third statement, and so on. A single statement is enough to get a response from JShell.

Fun fact: A tool like JShell, which responds immediately to each statement the programmer types, is called a *Read Evaluate Print Loop* (REPL). Many programming languages have REPLs. JShell is the "official" REPL of Java.

Instructions for launching JShell differ from one computer to the next. For some tips to help you run JShell on your computer, visit this book's website (`http://JavaForDummies.allmycode.com`).

WARNING

Some folks have tweaked JShell to make its behavior a bit different from what I describe in this book. If you're running IntelliJ IDEA's JShell Console or some other specialized JShell variant, be sure to check the vendor's documentation.

In Figure 4-7, I use JShell to find out how Java responds to the assignment statements in Listings 4-2 and 4-3.

```
jshell> double amountInAccount
amountInAccount ==> 0.0

jshell> amountInAccount = 50.22
amountInAccount ==> 50.22

jshell> amountInAccount = amountInAccount + 1000000.00
amountInAccount ==> 1000050.22

jshell> int weightOfAPerson, elevatorWeightLimit
weightOfAPerson ==> 0
elevatorWeightLimit ==> 0

jshell> weightOfAPerson = 150;
weightOfAPerson ==> 150

jshell> elevatorWeightLimit = 1400
elevatorWeightLimit ==> 1400

jshell> elevatorWeightLimit / weightOfAPerson
$8 ==> 9

jshell> $8 + 1
$9 ==> 10

jshell> 42 + 7
$10 ==> 49

jshell>
```

FIGURE 4-7: An intimate conversation between JShell and me.

When you run JShell, the dialogue goes something like this:

```
jshell> You type a statement
JShell responds

jshell> You type another statement
JShell responds
```

For example, in Figure 4-7, I type `double amountInAccount` and then press Enter. JShell responds by displaying

```
amountInAccount ==> 0.0
```

Here are a few things to notice about JShell:

- **You don't have to type an entire Java program.**

  Typing a few statements such as

  ```
  double amountInAccount
  amountInAccount = 50.22
  amountInAccount = amountInAccount + 1000000.00
  ```

does the trick. It's like running the code snippet in Listing 4-1 (except that Listing 4-1 doesn't declare `amountInAccount` to be a `double`).

- **In JShell, semicolons are (to a large extent) optional.**

  In Figure 4-7, I type a semicolon at the end of only one of my nine lines.

  For some advice about using semicolons in JShell, see Chapter 5.

CROSS REFERENCE

- **JShell responds immediately after you type each line.**

  After I declare `amountInAccount` to be `double`, JShell responds by telling me that the `amountInAccount` variable has the value `0.0`. After I type `amountInAccount = amountInAccount + 1000000.00`, JShell tells me that the new value of `amountInAccount` is `1000050.22`.

- **You can mix statements from many different Java programs.**

  In Figure 4-7, I mix statements from the programs in Listings 4-2 and 4-3. JShell doesn't care.

- **You can ask JShell for the value of an expression.**

  You don't have to assign the expression's value to a variable. For example, in Figure 4-7, I type

```
elevatorWeightLimit / weightOfAPerson
```

  JShell responds by telling me that the value of `elevatorWeightLimit / weightOfAPerson` is `9`. JShell makes up a temporary name for that value. In Figure 4-7, the name happens to be `$8`. So, on the next line in Figure 4-7, I ask for the value of `$8 +1`, and JShell gives me the answer `10`.

- **You can even get answers from JShell without using variables.**

  On the last line in Figure 4-7, I ask for the value of `42 + 7`, and JShell generously answers with the value `49`.

TIP

While you're running JShell, you don't have to retype commands that you've already typed. You don't even have to copy and paste commands. If you press the up-arrow key once, JShell shows you the command that you typed most recently. If you press the up-arrow key twice, JShell shows you the next-to-last command that you typed. And so on. When JShell shows you a command, you can press the left- and right-arrow keys to move to any character in the middle of the command. You can modify characters in the command. Finally, when you press Enter, JShell executes the newly modified command.

To end your run of JShell, you type **/exit** (starting with a slash). But `/exit` is only one of many commands you can give to JShell. To ask JShell what other kinds of commands you can use, type **/help**.

With JShell, you can test your statements before you put them into a full-blown Java program. That makes JShell a truly useful tool.

TIP

There are many ways to run JShell. If you can't get it running on your own computer, visit `https://tryjshell.org/` or `https://onecompiler.com/jshell/` to run JShell in your web browser. For a user-friendly interface to JShell, you can install Java and the JShell Easy extension in Visual Studio Code. If you're familiar with Python's Jupyter notebook environment, you can visit `https://github.com/dflib/jjava` and download the JJava kernel for JShell. (See Figure 4-8.)

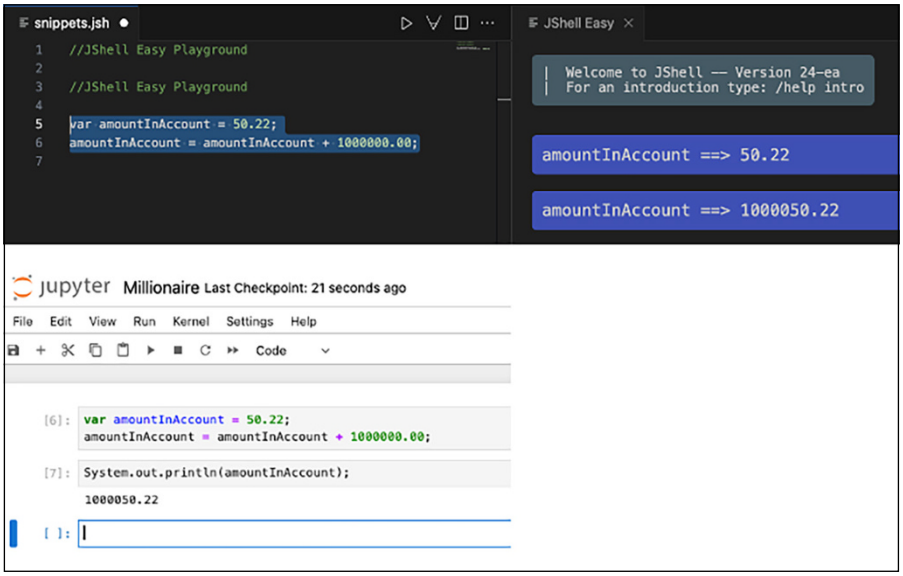


FIGURE 4-8: JShell Easy and the JJava Jupyter kernel.

SHELL GAME

TRY IT OUT

Visit this book's website (`http://JavaForDummies.allmycode.com`) for instructions on launching JShell on your computer. After launching JShell, type a few lines of code from Figure 4-7. See what happens when you type some slightly different lines.

The Atoms: Java's Primitive Types

The words *int* and *double* that I describe in the previous sections are examples of *primitive types* (also known as *simple* types) in Java. The Java language has exactly eight primitive types. As a newcomer to Java, you can pretty much ignore all but

four of these types. (As programming languages go, Java is nice and compact that way.) Table 4-1 shows the complete list of primitive types.

TABLE 4-1 Java's Primitive Types

Type Name	What a Literal Looks Like	Range of Values
Whole number types		
`byte`	`(byte)42`	–128 to 127
`short`	`(short)42`	–32768 to 32767
`int`	`42`	–2147483648 to 2147483647
`long`	`42L`	–9223372036854775808 to 9223372036854775807
Decimal number types		
`float`	`42.0F`	-3.4×10^{38} to 3.4×10^{38}
`double`	`42.0`	-1.8×10^{308} to 1.8×10^{308}
Character type		
`char`	`'A'`	Thousands of characters, glyphs, and symbols
Logical type		
`boolean`	`true`	true, false

The types that you shouldn't ignore are `int`, `double`, `char`, and `boolean`. Previous sections in this chapter cover the `int` and `double` types. So the next two sections cover `char` and `boolean` types.

The char type

What in the world is a Jhlzhy Jpwoly?

In the Roman (English) alphabet, the letter *J* comes seven letters after the letter *C*, and *h* comes seven letters after *a*. If you do this 7-letter trick with all the letters in *Jhlzhy Jpwoly*, you spell *Caesar Cipher*. It's the way Roman general Julius encrypted his secret messages.

The Java type that's used to store characters is called *char.* The program in Listing 4-4 uses Java's `char` type to find the letter in the alphabet that comes seven places before *J*. The code interprets only one letter, and it wouldn't work if that letter came before *H* in the alphabet. But let's not worry about that now.

LISTING 4-4: Seven Letters Before *J*

```
void main() {
    char oneLetter = 'J' - 7;
    println(oneLetter);
}
```

Figure 4-9 shows the program's output.

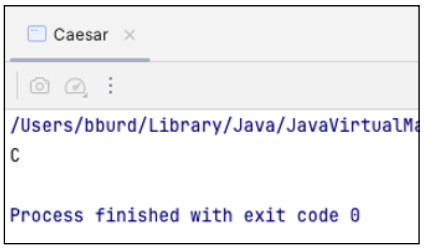


FIGURE 4-9: An exciting run of the program of Listing 4-4 as it appears in IntelliJ's Run tool window.

In Listing 4-4, notice how *J* is surrounded by single quote marks. In Java, every `char` literal starts and ends with a single quote mark.

REMEMBER

In a Java program, single quote marks surround the letter in a `char` literal.

In the section entitled "The types of values that variables may have," it says that the bit sequence 01001010 stands for the letter *J*, the number 74, or $1.0369608636003646 \times 10^{-43}$ depending on how you want to interpret it. Table 4-2 shows you what some other bit sequences may represent.

TABLE 4-2 How Java Represents Certain Characters

Sequence of Bits	What the Sequence Means When We Interpret it as a `char`	What the Sequence Means When We Interpret it as an `int`
1000001	`'A'`	65
1000010	`'B'`	66
1000011	`'C'`	67
1000100	`'D'`	68
1000101	`'E'`	69
1000110	`'F'`	70

Sequence of Bits	What the Sequence Means When We Interpret it as a `char`	What the Sequence Means When We Interpret it as an `int`
1000111	`'G'`	71
1001000	`'H'`	72
1001001	`'I'`	73
1001010	`'J'`	74
...	...	...
1100001	`'a'`	97

In many ways, Java treats `char` as a numeric type. In fact, the literal `'J'` really stands for the number 74. The fact that this literal can look like a letter on the computer screen is just Java's way of making you feel as if you're dealing with symbols in the alphabet. So, in Listing 4-4, when you subtract 7 from `'J'`, you get the numeric value 67. And, when you call `println`, that number appears on the screen as the letter *C*.

WARNING

If you're tempted to write the following statement,

```
char myLittleChars = 'barry'; //Don't do this
```

please resist the temptation. You can't store more than one letter at a time in a `char` variable, and you can't put more than one letter between a pair of single quotes. If you're trying to store words or sentences (not just single letters), you need to use something called a `String`.

For a look at Java's `String` type, see the section "The Molecules and Compounds: Reference Types," later in this chapter.

WARNING

Java's handling of characters has some subtle twists and turns. For example, in Listing 4-4, you've declared `oneLetter` to be of type `char`. So, the statement `println(oneLetter)` displays the letter `C`. But if your program contains the statement `println('J' - 7)`, Java thinks you're asking it to subtract 7 from 74, so the output you see is the number `67`.

TECHNICAL STUFF

If you're used to writing programs in other languages, you may be aware of something called ASCII character encoding. Many languages use ASCII; Java uses Unicode. In the old ASCII representation, each character takes up only 8 bits, but in Unicode, each character takes up 8, 16, or 32 bits. Whereas ASCII stores the letters of the Roman alphabet, Unicode has room for characters from many of the world's languages. Java's `char` type is 16-bit Unicode. So, in Table 4-2, the sequence 1001010 actually stands for 000000001001010.

The boolean type

A variable of type `boolean` stores one of two values: `true` or `false`. Listing 4-5 demonstrates the use of a `boolean` variable. Figure 4-10 shows the output of the program in Listing 4-5.

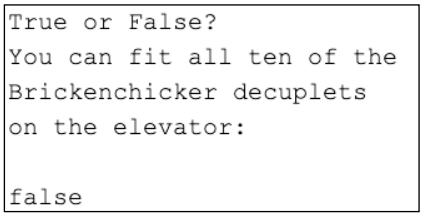


FIGURE 4-10: The Brickenchicker decuplets strike again.

LISTING 4-5: **Using the boolean Type**

```
void main() {
    println("True or False?");
    println("You can fit all ten of the");
    println("Brickenchicker decuplets");
    println("on the elevator:");
    println("");  // or println();

    int weightOfAPerson = 150;
    int elevatorWeightLimit = 1400;
    int numberOfPeople = elevatorWeightLimit / weightOfAPerson;

    boolean allTenOkay = numberOfPeople >= 10;

    println(allTenOkay);
}
```

In Listing 4-5, the `allTenOkay` variable is of type `boolean`. To find a value for the `allTenOkay` variable, the program checks to see whether `numberOfPeople` is greater than or equal to `ten`. (The symbols `>=` stand for *greater than or equal to.*)

At this point, it pays to be fussy about terminology. Any part of a Java program that has a value is an *expression.* If you write

```
weightOfAPerson = 150;
```

then `150` is an expression (an expression whose value is the quantity `150`). If you write

```
numberOfEggs = 2 + 2;
```

then `2 + 2` is an expression (because 2 + 2 has the value `4`). If you write

```
int numberOfPeople = elevatorWeightLimit / weightOfAPerson;
```

then `elevatorWeightLimit / weightOfAPerson` is an expression. (The value of the expression `elevatorWeightLimit / weightOfAPerson` depends on whatever values the variables `elevatorWeightLimit` and `weightOfAPerson` have when the code containing the expression is executed.)

REMEMBER

Any part of a Java program that has a value is an *expression.*

In Listing 4-5, the code `numberOfPeople >= 10` is an expression. The expression's value depends on the value stored in the `numberOfPeople` variable. But, as you know from seeing the strawberry shortcake at the Brickenchicker family's catered lunch, the value of `numberOfPeople` isn't greater than or equal to ten. As a result, the value of `numberOfPeople >= 10` is `false`. So, in the statement in Listing 4-5, in which `allTenOkay` is assigned a value, the `allTenOkay` variable is assigned a `false` value.

In Listing 4-5, the call `println("")` adds a blank line to the program's output. That's because `""` represents the empty string — the string that doesn't contain any characters. For more information about strings, see this chapter's "Java's String type" section.

TECHNICAL STUFF

With the preview features in Java 23, you can display a blank line by calling `System.out.println()`, but not by calling `println()` alone. By the time you read this book, a simple `println()` call will probably be available. In fact, the masters of Java plan to add that simple `println()` in Java 24. I don't have a time machine, so I can't try that feature before I send this chapter to Wiley.

Creating New Values by Applying Operators

What could be more comforting than your old friend the plus sign? It was the first topic you learned about in elementary school math. Almost everybody knows how to add 2 and 2. In fact, in English usage, adding 2 and 2 is a metaphor for

something that's easy to do. Whenever you see a plus sign, a cell in your brain says, "Thank goodness — it could be something much more complicated."

Java has a plus sign. You can use it for several purposes. You can use the plus sign to add two numbers, like this:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
```

You can also use the plus sign to paste `String` values together:

```
String startOfChapter =
    "It's three in the morning. I'm dreaming about the" +
    "history course that I failed in high school.";
println(startOfChapter);
```

This can be handy because, in Java, you can't make an ordinary quoted `String` literal straddle from one line to another. In other words, the following code wouldn't work:

```
String thisIsBadCode =
    "It's three in the morning. I'm dreaming about the
    history course that I failed in high school.";
println(thisIsBadCode);
```

If you want a single string to span several lines, you need another trick. For details, see Chapter 5.

The correct way to say that you're pasting `String` values together is to say that you're *concatenating* `String` values.

You can even use the plus sign to paste numbers next to `String` values:

```
int apples, oranges, fruit;
apples = 5;
oranges = 16;
fruit = apples + oranges;
println("You have" + fruit + "pieces of fruit.");
```

Of course, the old minus sign is available, too (but not for `String` values):

```
apples = fruit - oranges;
```

Use an asterisk (*) for multiplication and a slash (/) for division:

```
double rate, pay;
int hours;

rate = 6.25;
hours = 35;
pay = rate * hours;
println(pay);
```

For an example using division, refer to Listing 4-3.

WARNING

When you divide an `int` value by another `int` value, you get an `int` value. The computer doesn't round. Instead, the computer chops off any remainder. If you put `println(11 / 4)` in your program, the computer prints `2`, not `2.75`. To get past this, make either (or both) of the numbers you're dividing `double` values. If you put `println(11.0 / 4)` in your program, the computer prints `2.75`.

Another useful arithmetic operator is called the *remainder* operator. The symbol for the remainder operator is the percent sign (`%`). When you put `println(11 % 4)` in your program, the computer prints `3`. It does this because 4 goes into 11 who-cares-how-many times with a remainder of 3. The remainder operator turns out to be fairly useful. Listing 4-6 has an example.

LISTING 4-6: **Making Change**

```
void main() {
    int total = 248;
    int quarters = total / 25;
    int whatsLeft = total % 25;

    int dimes = whatsLeft / 10;
    whatsLeft = whatsLeft % 10;

    int nickels = whatsLeft / 5;
    whatsLeft = whatsLeft % 5;

    int cents = whatsLeft;

    println("From " + total + " cents you get");
    println(quarters + " quarters");
    println(dimes + " dimes");
    println(nickels + " nickels");
    println(cents + " cents");
}
```

Figure 4-11 shows a run of the code in Listing 4-6. You start with a total of 248 cents. Then

```
quarters = total / 25
```

divides 248 by 25, giving 9. That means you can make 9 quarters from 248 cents. Next,

```
whatsLeft = total % 25
```

divides 248 by 25 again and puts only the remainder, 23, into `whatsLeft`. Now you're ready for the next step, which is to take as many dimes as you can out of cents.

```
From 248 cents you get
9 quarters
2 dimes
0 nickels
3 cents
```

FIGURE 4-11: Change for $2.48.

TECHNICAL STUFF

The code in Listing 4-6 makes change in US currency with the following coin denominations: 1 cent, 5 cents (one nickel), 10 cents (one dime), and 25 cents (one quarter). In fact, Listing 4-6 gives you the *smallest number of coins* that add up to 248 cents. With some minor tweaking, you can make the code work in any country's coinage. You can always get a set of coins adding up to a total. But, for some kinds of coinage, you won't always get the *smallest number of coins* that add up to a total. For example, in the mid-1970s, England had coins with values 25p, 20p, 10p, and 5p. To put 40p together, a program like the one in Listing 4-6 would suggest 25p + 10p + 5p. But you could use fewer coins by shelling out 20p + 20p.

TRY IT OUT

SMOOTH OPERATORS

Find the values of the following expressions by typing each expression in JShell:

```
» 5 / 4
» 5 / 4.0
» 5.0 / 4
» 5.0 / 4.0
» "5" + "4"
» 5 + 4
» " " + 5 + 4
```

Initialize once, assign often

Listing 4-6 has three lines that put values into the variable `whatsLeft`:

```
int whatsLeft = total % 25;

whatsLeft = whatsLeft % 10;

whatsLeft = whatsLeft % 5;
```

Only one of these lines is a declaration. The other two lines are assignment statements. That's good because you can't declare the same variable more than once (not without creating something called a *block*). If you goof and write

```
int whatsLeft = total % 25;

int whatsLeft = whatsLeft % 10;
```

in Listing 4-6, you see an error message (such as `Duplicate variable whatsLeft` or `Variable 'whatsLeft' is already defined`) when you try to compile your code.

CROSS REFERENCE

To find out what a block is, see Chapter 5. Then, for some honest talk about redeclaring variables, see Chapter 7.

The increment and decrement operators

Java has some neat little operators that make life easier (for the computer's processor, for your brain, and for your fingers). Altogether, four such operators exist — two increment operators and two decrement operators. The increment operators add 1, and the decrement operators subtract 1. The increment operators use double plus signs (++), and the decrement operators use double minus signs (--). To see how they work, you need some examples. The first example is shown in Figure 4-12.

Figure 4-13 shows a run of the program in Figure 4-12. In this horribly uneventful run, the count of bunnies prints three times.

The double plus signs go by two names, depending on where you put them. When you put the ++ before a variable, the ++ is called the *preincrement* operator. (The *pre* stands for *before.*)

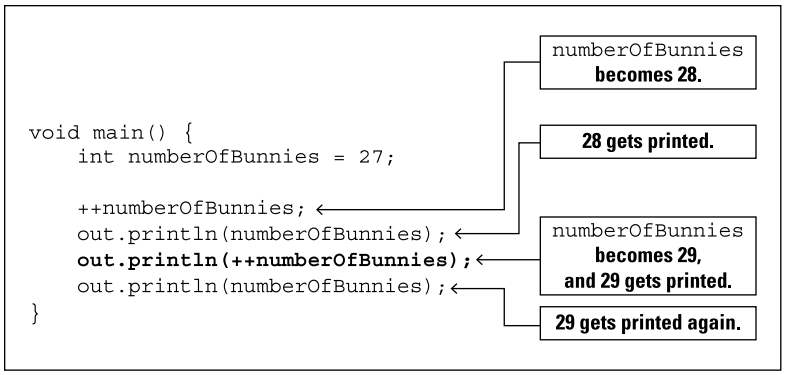


FIGURE 4-12: Using preincrement.

```
28
29
29
```

FIGURE 4-13: A run of the code in Figure 4-12.

The word *before* has two meanings:

- You put ++ before the variable.
- The computer adds 1 to the variable's value before the variable is used in any other part of the statement.

To understand this, look at the bold line in Figure 4-12. The computer adds 1 to `numberOfBunnies` (raising the value of `numberOfBunnies` to 29) and then prints 29 onscreen.

REMEMBER

With `println(++numberOfBunnies)`, the computer adds 1 to `numberOfBunnies` before printing the new value of `numberOfBunnies` onscreen.

An alternative to preincrement is *postincrement.* (The *post* stands for *after.*) The word *after* has two different meanings:

- You put ++ after the variable.
- The computer adds 1 to the variable's value after the variable is used in any other part of the statement.

To see more clearly how postincrement works, look at the bold line in Figure 4-14. The computer prints the old value of `numberOfBunnies` (which is 28) on the screen, and then the computer adds 1 to `numberOfBunnies`, which raises the value of `numberOfBunnies` to 29.

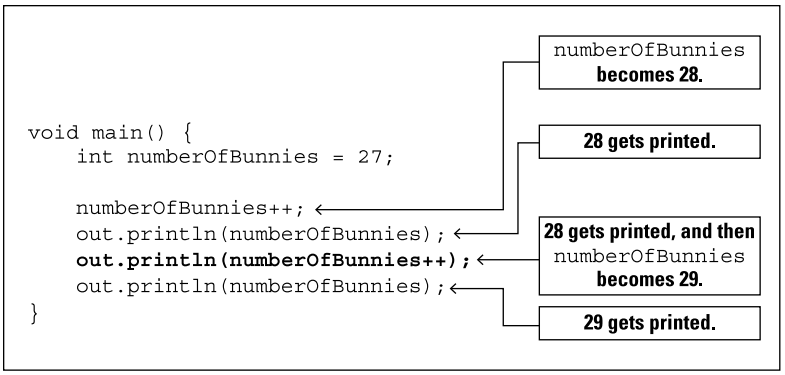


FIGURE 4-14: Using postincrement.

REMEMBER

With `println(numberOfBunnies++)`, the computer adds 1 to `numberOfBunnies` after printing the old value that `numberOfBunnies` already had.

Figure 4-15 shows a run of the code in Figure 4-14. Compare Figure 4-15 with the run in Figure 4-13:

- With preincrement in Figure 4-13, the second number is 29.
- With postincrement in Figure 4-15, the second number is 28.

  In Figure 4-15, 29 doesn't show onscreen until the end of the run, when the computer executes one last `println(numberOfBunnies)`.

FIGURE 4-15: A run of the code in Figure 4-14.

TIP

Are you trying to decide between using preincrement or postincrement? Try no longer. Most programmers use postincrement. In a typical Java program, you often see things like `numberOfBunnies++`. You seldom see things like `++numberOfBunnies`.

In addition to preincrement and postincrement, Java has two operators that use `--`. These operators are called *predecrement* and *postdecrement:*

- With **predecrement** (`--numberOfBunnies`), the computer subtracts 1 from the variable's value before the variable is used in the rest of the statement.
- With **postdecrement** (`numberOfBunnies--`), the computer subtracts 1 from the variable's value after the variable is used in the rest of the statement.

STATEMENTS AND EXPRESSIONS

You can describe the pre- and postincrement and pre- and postdecrement operators in two ways: the way everyone understands them and the right way. The way that I explain the concept in most of this section (in terms of time, with *before* and *after*) is the way that everyone understands it. Unfortunately, the way everyone understands the concept isn't really the right way. When you see ++ or --, you can think in terms of time sequence. But occasionally a programmer uses ++ or -- in a convoluted way, and the notions of *before* and *after* break down. So, if you're ever in a tight spot, think about these operators in terms of statements and expressions.

First, remember that a statement tells the computer to do something, and an expression has a value. (I discuss statements in Chapter 3, and I describe expressions elsewhere in this chapter.) Which category does `numberOfBunnies++` belong to? The surprising answer is both — the Java code `numberOfBunnies++` is both a statement and an expression.

Assume that, before the computer executes the code `println(numberOfBunnies++)`, the value of `numberOfBunnies` is 28:

- As a statement, `numberOfBunnies++` tells the computer to add 1 to `numberOfBunnies`.
- As an expression, the value of `numberOfBunnies++` is 28, not 29.

So, even though the computer adds 1 to `numberOfBunnies`, the code `println(numberOfBunnies++)` really means `println(28)`.

Now, almost everything you just read about `numberOfBunnies++` is true about `++numberOfBunnies`. The only difference is that as an expression, `++numberOfBunnies` behaves in a more intuitive way:

- As a statement, `++numberOfBunnies` tells the computer to add 1 to `numberOfBunnies`.
- As an expression, the value of `++numberOfBunnies` is 29.

So, with `println(++numberOfBunnies)`, the computer adds 1 to the variable `numberOfBunnies`, and the code `println(++numberOfBunnies)` really means `println(29)`.

TECHNICAL STUFF

Rather than write `++numberOfBunnies`, you can achieve the same effect by writing `numberOfBunnies = numberOfBunnies + 1`. So, some people conclude that Java's `++` and `--` operators are for saving keystrokes — to keep those poor fingers from overworking themselves. This is entirely incorrect. The best reason for using `++` is to avoid the inefficient and error-prone practice of writing the same variable name, such as `numberOfBunnies`, twice in the same statement. If you write `numberOfBunnies` only once (as you do when you use `++` or `--`), the computer has to figure out what `numberOfBunnies` means only once. On top of that, when you write `numberOfBunnies` only once, you have only one chance (instead of two chances) to type the variable name incorrectly. With simple expressions like `numberOfBunnies++`, these advantages hardly make a difference. But with more complicated expressions, such as `inventoryItems[(quantityReceived--*itemsPerBox+17)]++`, the efficiency and accuracy that you gain by using `++` and `--` are significant.

PROGNOSTICATION GAME

TRY IT OUT

Before you run the following code, try to predict what the code's output will be. Then run the code to find out whether your prediction is correct:

```
void main() {
    int i = 10;
    println(i++);
    println(--i);
    --i;
    i--;
    println(i);
    println(++i);
    println(i--);
    println(i);
    i++;
    i = i++ + ++i;
    println(i);
    i = i++ + i++;
    println(i);
}
```

SEE PLUS PLUS

Type the boldface text, one line after another, into JShell and see how JShell responds:

```
int i = 8
i++
```

```
i
i
i++
i
++i
i + i++
i++ + i
```

Assignment operators

If you read the preceding section, which is about operators that add 1, you may be wondering whether you can manipulate these operators to add 2 or add 5 or add 1000000. Can you write `numberOfBunnies++++` and still call yourself a Java programmer? Well, you can't. If you try it, an error message appears when you try to compile your code.

What can you do? As luck would have it, Java has plenty of assignment operators you can use. With an *assignment operator,* you can add, subtract, multiply, or divide by anything you want. You can do other cool operations, too. Listing 4-7 has a smorgasbord of assignment operators (the ones with equal signs). Figure 4-16 shows the output from running Listing 4-7.

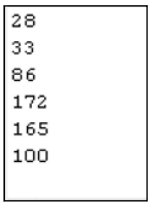


FIGURE 4-16: A run of the code in Listing 4-7.

LISTING 4-7: **Assignment Operators**

```
void main() {
    int numberOfBunnies = 27;
    int numberExtra = 53;

    numberOfBunnies += 1;
    println(numberOfBunnies);

    numberOfBunnies += 5;
    println(numberOfBunnies);
```

```
        numberOfBunnies += numberExtra;
        println(numberOfBunnies);

        numberOfBunnies *= 2;
        println(numberOfBunnies);

        println(numberOfBunnies -= 7);

        println(numberOfBunnies = 100);
    }
```

Listing 4-7 shows how versatile Java's assignment operators are. With the assignment operators, you can add, subtract, multiply, or divide a variable by any number. Notice how `+= 5` adds 5 to `numberOfBunnies`, and how `*= 2` multiplies `numberOfBunnies` by `2`. You can even use another expression's value (in Listing 4-7, `numberExtra`) as the number to be applied.

The last two lines in Listing 4-7 demonstrate a special feature of Java's assignment operators. You can use an assignment operator as part of a larger Java statement. In the next-to-last line of Listing 4-7, the operator subtracts 7 from `numberOfBunnies`, decreasing the value of `numberOfBunnies` from `172` to `165`. Then the whole assignment business is stuffed into a call to `println`, so `165` prints onscreen.

Lo and behold, the last line of Listing 4-7 shows how you can do the same thing with Java's plain old equal sign. The thing that I call an assignment statement near the start of this chapter is really one of the assignment operators that I describe in this section. Therefore, whenever you assign a value to something, you can make that assignment be part of a larger statement.

TECHNICAL STUFF

Each use of an assignment operator does double duty as a statement and an expression. In all cases, the expression's value equals whatever value you assign. For example, before executing the code `println(numberOfBunnies -= 7)`, the value of `numberOfBunnies` is `172`. As a statement, `numberOfBunnies -= 7` tells the computer to subtract 7 from `numberOfBunnies` (so the value of `numberOfBunnies` goes from `172` to `165`). As an expression, the value of `numberOfBunnies -= 7` is `165`. So the code `println(numberOfBunnies -= 7)` really means `println(165)`. The number `165` displays on the computer screen.

For a richer explanation of this kind of thing, see the sidebar "Statements and expressions," earlier in this chapter.

THE OPERATION IS A SUCCESS

TRY IT OUT

Before you run the following code, try to predict what the code's output will be. Then run the code to find out whether your prediction is correct:

```
void main() {
    int i = 10;
    i += 2;
    i -= 5;
    i *= 6;
    println(i);
    println(i += 3);
    println(i /= 2);
}
```

The Molecules and Compounds: Reference Types

By combining simple things, you get more complicated things. That's the way things always go. Take some of Java's primitive types, whip them together to make a primitive type stew, and what do you get? You get a more complicated type called a *reference type.*

The program in Listing 4-8 uses reference types. Figure 4-17 shows you a run of the program in Listing 4-8.

```
NE ER GI E A SUCKER AN E EN BREAK
You can win $550
```

FIGURE 4-17: A word puzzle and a wheel spin.

LISTING 4-8: Using Reference Types

```
void main() {
    String solvedPuzzle = "NEVER GIVE A SUCKER AN EVEN BREAK";
    String unsolvedPuzzle = solvedPuzzle.replace('V', ' ');
    println(unsolvedPuzzle);

    Random myRandom;
    myRandom = new Random();
    int number = 500 + 50 * myRandom.nextInt(9);
    println("You can win $" + number);
}
```

What use would anyone have for the code in Listing 4-8? Picture this: On a certain TV game show, contestants solve a puzzle by guessing which letters are missing from a well-known phrase or sentence. They spin a wheel that can stop on any of the numbers 500, 550, 600, . . ., up to 900. A number on the wheel is the amount of money a contestant may win.

The program in Listing 4-8 uses two reference types – *String* and *Random*. Both types are defined in the Java API.

Java's String type

Roughly speaking, an instance of the `String` class is a bunch of characters. It's like having several `char` values in a row. Here are some examples:

- `"x% z "` is a `String` literal containing five characters, two of which are blank spaces.
- `""` is the empty string – the `String` literal that doesn't contain any characters.
- `"A"` is a `String` literal that contains only one character. Despite what some people think, this isn't the same as the `char` literal `'A'`. It's important to remember this because there are things you can do with `String` values that you can't do with `char` values, and vice-versa.

Java's Random type

An instance of the `Random` class is a formula that can pump out numbers that appear to be randomly generated (although they're not *really* randomly generated). Figure 4-18 shows you why the expression `500 + 50 * myRandom.nextInt(9)` generates one of the numbers 500, 550, 600, . . ., up to 900.

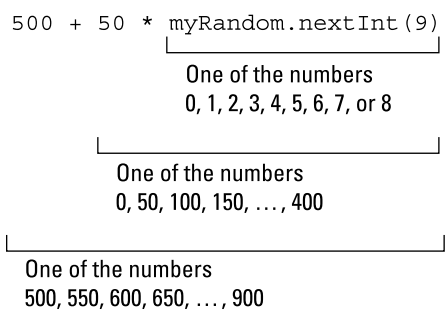

500 + 50 * myRandom.nextInt(9)
One of the numbers
0, 1, 2, 3, 4, 5, 6, 7, or 8
One of the numbers
0, 50, 100, 150, . . . , 400
One of the numbers
500, 550, 600, 650, . . . , 900

FIGURE 4-18: Spinning the wheel.

Coding with classes, objects, and their methods

You can look up `String` and `Random` in Java's API documentation. But, even before you do, I can tell you what you'll find. You'll find that `String` and `Random` are the names of Java classes. So that's the big news. Every class is the name of a reference type.

CROSS REFERENCE

To review the notion of a Java class, see the section on object-oriented programming (OOP) in Chapter 1. And, to read more about Java classes, see Chapter 7.

In Listing 4-8, I do four things with the classes `String` and `Random`:

- I reserve variable names to refer to instances of `String` and `Random`.
- I create instances of `String` and `Random`.
- I make the variables refer to the instances.
- I call a method on each of the instances.

Reserving a variable name

In Listing 4-2, I reserve the variable name `amountInAccount` for storing `double` values by writing `double amountInAccount`. In the same way, in Listing 4-8, I reserve `myRandom` for referring to a `Random` instance by writing `Random myRandom`. (See Item 1 in Figure 4-19.)

Reserving a variable name

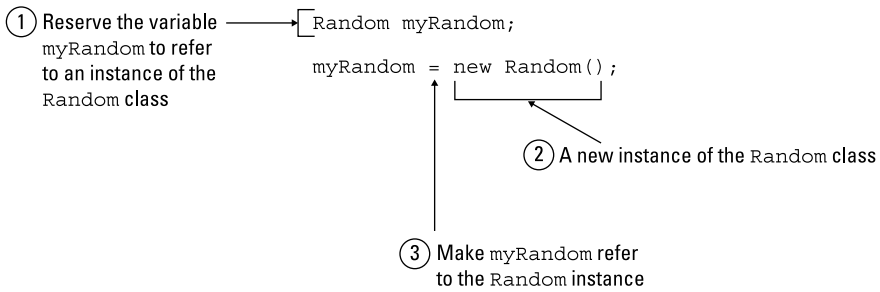

1 Reserve the variable myRandom to refer to an instance of the Random class
Random myRandom;
myRandom = new Random();
2 A new instance of the Random class
3 Make myRandom refer to the Random instance

FIGURE 4-19: Making `myRandom`.

Creating an instance and making a variable refer to the instance

In Listing 4-8, I create an instance of the `Random` class by writing `new Random()`. The statement

```
myRandom = new Random();
```

makes the variable `myRandom` refer to that newly-created instance. (See Items 2 and 3 in Figure 4-19.) In Listing 4-8, I could have written one statement instead of two:

```
Random myRandom = new Random();
```

This single statement would do double duty by reserving the name `myRandom` and making `myRandom` refer to a newly-created object. I could even have written

```
var Random myRandom = new Random();
```

and let Java figure out the variable's type.

REMEMBER

When you use Java's `new` keyword, you create a brand-new object.

Figure 4-20 illustrates the same business for the `solvedPuzzle` variable in Listing 4-8.

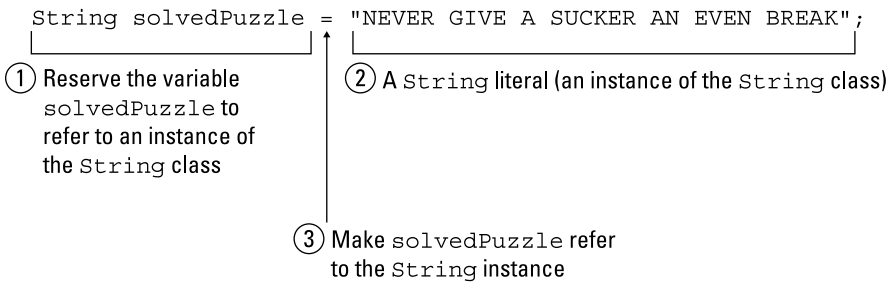

String solvedPuzzle = "NEVER GIVE A SUCKER AN EVEN BREAK";
1 Reserve the variable solvedPuzzle to refer to an instance of the String class
2 A String literal (an instance of the String class)
3 Make solvedPuzzle refer to the String instance

FIGURE 4-20: Making `solvedPuzzle`.

For the `solvedPuzzle` variable, I could have written

```
String solvedPuzzle = new String("NEVER GIVE A SUCKER AN EVEN BREAK");
```

But a double-quoted `String` literal is already an instance of the `String` class. So, with strings, you can usually omit the `new String` part.

Calling a method

If you refer back to Chapter 1, you see that each class describes both data and code for working with the data. In Listing 4-8, the `String` instance has data (the characters `"NEVER GIVE A SUCKER AN EVEN BREAK"`). But it also has code for working with the data. Each such piece of code is actually a method.

CROSS REFERENCE

For an introduction to methods, refer to Chapter 3.

In Java's API, the `String` class has its own method named `replace`. When you write `solvedPuzzle.replace('V', ' ')`, you're asking Java to call the `solvedPuzzle` instance's `replace` method. You're telling Java to create a new string by replacing each occurrence of the letter `'V'` in `solvedPuzzle` with a blank space. (See Figure 4-21.)

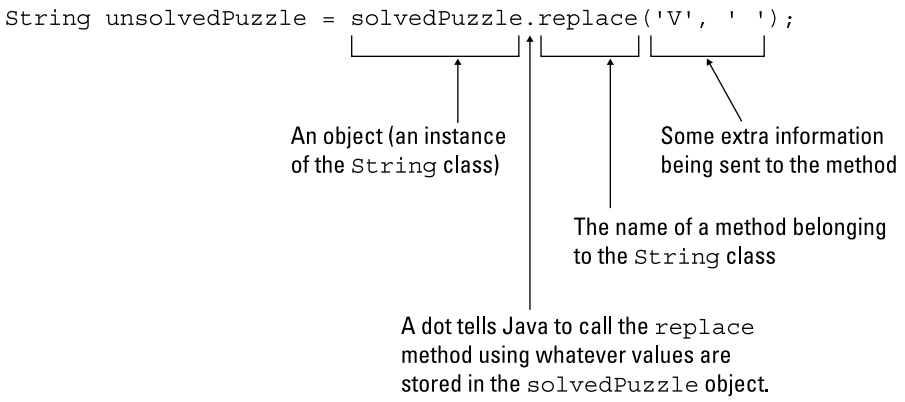

String unsolvedPuzzle = solvedPuzzle.replace('V', ' ');
An object (an instance of the String class)
Some extra information being sent to the method
The name of a method belonging to the String class
A dot tells Java to call the replace method using whatever values are stored in the solvedPuzzle object.

FIGURE 4-21: Calling the `String` class's `replace` method.

Also, in Java's API, the `Random` class has its own `nextInt` method. Remember that the `myRandom` object is a formula that can pump out numbers that appear to be randomly generated. When you write `myRandom.nextInt(9)`, you're asking Java to use the `myRandom` formula to feed you one of the numbers 0, 1, 2, 3, . . ., 8. (See Figure 4-22.)

In Listing 4-8, you call methods by writing `solvedPuzzle.replace('V', ' ')` and `myRandom.nextInt(9)`. Do you see the pattern? In general, you call an object's method by putting a dot between the object and the method. You write something like *`someObject.doSomething(someExtraInformation)`*.

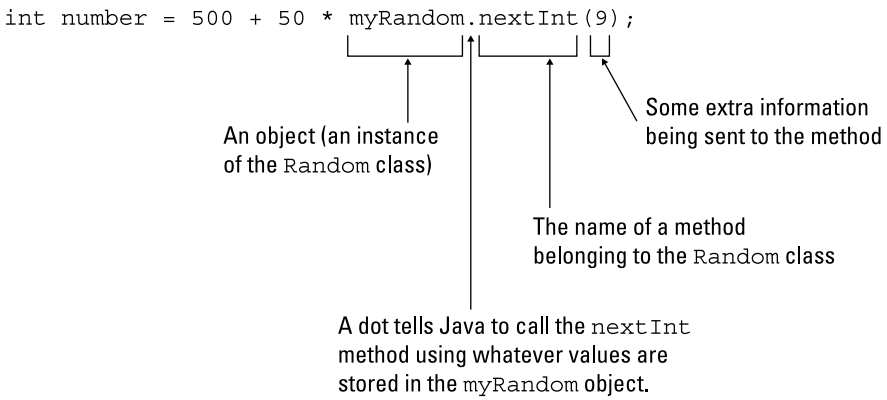

int number = 500 + 50 * myRandom.nextInt(9);
An object (an instance of the Random class)
Some extra information being sent to the method
The name of a method belonging to the Random class
A dot tells Java to call the nextInt method using whatever values are stored in the myRandom object.

FIGURE 4-22: Calling the `Random` class's `nextInt` method.

What did he just say?

This talk about OOP can be overwhelming, so to keep you from getting a throbbing headache, I summarize some main points:

- A class describes both data and code for working with the data. A piece of code to work with the data is a method.
- You can reserve a variable name so it can refer to an instance of a particular class. To do this, you write *`ClassName variableName`*.
- You can create an object – an instance of a particular class. To do this, you sometimes use the class's name preceded by Java's `new` keyword. Literals (like `"NEVER"`, `'J'`, and `47.92`) are already objects, so you can assign them to variables without using `new`.
- You can call a method belonging to an object. To do this, you typically put a dot between the object's name and the method's name.

TRY IT OUT

Try these experiments:

RUN IT AGAIN AND AGAIN

Run the code in Listing 4-8 a few times on your computer. Does the winning amount seem to be randomly generated?

PLANT A SEED

First, run the following code a few times and make some notes so you can keep track of the results.

```
Random yourRandom;
yourRandom = new Random();
```

```
println(yourRandom.nextInt(100));
println(yourRandom.nextInt(100));
println(yourRandom.nextInt(100));
```

Next modify the code by changing `new Random()` to `new Random(12345)`. Run this modified code a few times. How do the results differ from the those in the first set of runs?

SAGGING STRINGS

The first statement in Listing 4-8 is the declaration of the `solvedPuzzle` variable. Immediately after that statement, add a call to the `String` class's `toLowerCase` method:

```
solvedPuzzle = solvedPuzzle.toLowerCase();
```

What happens when you run this code?

I'D RATHER BE IN PHILADELPHIA

The first statement in Listing 4-8 is the declaration of the `solvedPuzzle` variable. Immediately after that statement, add a call to one of the `String` class's `replace` methods:

```
solvedPuzzle = solvedPuzzle.replace("NEVER", "ALWAYS");
```

What happens when you run this code? Look online for a Javadoc page describing the `String` class and find out how many methods the `String` class has.

IN THIS CHAPTER

- Writing statements that choose between alternatives
- Forming logical conditions
- Putting statements inside one another
- Choosing among many alternatives

Chapter 5

Controlling the Flow by Making Decisions

The TV show *Dennis the Menace* aired on CBS from 1959 to 1963. I remember one episode in which Mr. Wilson was having trouble making an important decision. I think it was something about changing jobs or moving to a new town. Anyway, I can still see that shot of Mr. Wilson sitting in his yard, sipping lemonade, and staring into nowhere for the whole afternoon. Of course, the annoying character Dennis was continually interrupting Mr. Wilson's peace and quiet. That's what made this situation funny.

What impressed me about this episode (the reason I remember it clearly, even now) was Mr. Wilson's dogged intent in making the decision. This guy wasn't going about his everyday business, roaming around the neighborhood while thoughts about the decision wandered in and out of his mind. He was sitting quietly in his yard, making marks carefully and logically on his mental balance sheet. How many people actually make decisions this way?

At that time, I was still pretty young. I'd never faced the responsibility of making a big decision that affected my family and me. But I wondered what such a decision-making process would be like. Would it help to sit there like a stump for hours on end? Would I make my decisions by the careful weighing and tallying of options? Or would I shoot in the dark, take risks, and act on impulse? Only time would tell.

Making Decisions (Java if Statements)

When you're writing computer programs, you're constantly hitting forks in roads. Did the user correctly type the password? If yes, let the user work; if no, kick the bum out. So the Java programming language needs a way of making a program branch in one of two directions. Fortunately, the language has a way: It's called an `if` statement.

Guess the number

Listing 5-1 illustrates the use of an `if` statement. Two runs of the program in Listing 5-1 are shown in Figure 5-1.

```
Pick a number from 1 to 10: 10
You lose.
The random number was 7.
Thank you for playing.


Pick a number from 1 to 10: 2
**********
*You win.*
**********
Thank you for playing.
```

FIGURE 5-1: Two runs of the guessing game.

LISTING 5-1: A Guessing Game

```
void main() {
    String input = readln("Pick a number from 1 to 10: ");
    int inputNumber = Integer.parseInt(input);
    int randomNumber = new Random().nextInt(10) + 1;

    if (inputNumber == randomNumber) {
        println("**********");
        println("*You win.*");
        println("**********");
    } else {
        println("You lose.");
        print("The random number was ");
```

```
            println(randomNumber + ".");
        }

        println("Thank you for playing.");
    }
```

The program in Listing 5-1 plays a guessing game with the user. The program gets a number (a guess) from the user and then generates a random number between 1 and 10. If the number the user entered is the same as the random number, the user wins. Otherwise, the user loses and the program tells the user what the random number was.

She controlled keystrokes from the keyboard

In Listing 5-1, the line

```
String input = readln("Pick a number from 1 to 10: ");
```

does three things:

1. It displays the words `Pick a number from 1 to 10:` on the user's screen.
2. It waits for the user to type some characters and then press Enter.
3. After the user presses Enter, it makes the variable `input` refer to the characters that the user typed.

So, what happens on the first line in Figure 5-1? On that line, Java makes `input` refer to `"10"`.

Wait! At the end of that last sentence, why are there quotation marks around the characters `10`? The quotation marks indicate that, as its name suggests, the variable `input` refers to a string of characters. Java's `readln` method always interprets the user's input as a string of characters. That's why Listing 5-1 starts with the words `String input`.

The third statement in Listing 5-1 generates a random number from 1 to 10. (If you don't believe me, look at the section about Java's `Random` class in Chapter 4.) Anyway, the `Random` class's `nextInt` method always gives you an `int` value. Later on in the code, we have to compare two things: the stuff the user entered, which is a `String` instance, and the number that `Random` gave us, which is an `int` value.

ARGUING ABOUT JAVA

I've asked the stewards of Java about making a `readInt` method available. With `readInt`, you'd be able to write

```
int inputNumber = readInt("Pick a number from 1 to 10: ");
```

near the start of Listing 5-1, thus avoiding all the `Integer.parseInt` business until later in your Java journey. (I wrote about this possibility in an earlier version of Chapter 5.)

But, on this very day while I finish off Chapter 5, the stewards of Java have sent out an email saying, "Definitely not! No `readInt` method. End of story!" (Of course, I paraphrase.) So, I deleted all of this chapter's references to a possible `readInt` method.

Within a few hours of the initial email, several Java aficionados chimed in with emails about why we *should* have a `readInt` method. Now there's a vigorous discussion among these aficionados arguing for and against having a `readInt` method. So, I've added a few words about `readInt` again.

So, who knows? I see "No `readInt`," then "Yes `readInt`," then "No `readInt`," and then "Yes `readInt`." If `readInt` is available by the time you read this book then, by all means, use `readInt`!

In some languages, you can ask whether `input` equals `randomNumber`. But Java is fussy about such things. In Java, you can't compare `String` apples with `int` oranges. You have to compare strings with strings or `int` values with `int` values. So, in Listing 5-1, the line

```
int inputNumber = Integer.parseInt(input);
```

creates a new `int` variable named `inputNumber`. The method `Integer.parseInt` examines the string that we got from the user's input and interprets that string as an `int` value. Because `inputNumber` and `randomNumber` are both `int` values, we can compare them and find out whether they're equal to one another. Thank goodness for the `Integer.parseInt` method! It's defined in Java's API.

Reading double values: a cautionary tale

Chapter 4 describes Java's primitive types, including the `int` and `double` types. To read a `double` value from the keyboard, you mimic what you do in Listing 5-1 but you change a few words. Here's how it works:

```
void main() {
    String input = readln("Enter an amount: ");
    double inputNumber = Double.parseDouble(input);
    println(inputNumber + 0.5);
}
```

I receive plenty of emails from readers, and one of the most common questions is, "Why do I get an error message when I try to read a `double` value?" Decimal points vary from one country to another. In the United States, *10.5* (with a period) represents ten-and-a-half, but in France, *10,5* (with a comma) represents ten-and-a-half. In Switzerland, 10.50 is an amount of money, and 10,50 is an amount that's not money-related. In the Persian language, a decimal point looks like a slash (but sits a bit lower than the digit characters). Your computer's operating system stores information about the country you live in, and Java reads that information to decide what ten-and-a-half looks like. If you run a program containing a `Double.parseDouble` method call and Java responds with a `NumberFormatException`, check your input. You might have input 10.5 when your country's conventions require 10,5 (or another way of representing ten-and-a-half). For more information, see the sidebar "Where on earth do you live?" in Chapter 9.

The if statement

At the core of Listing 5-1 is a Java `if` statement. This `if` statement represents a fork in the road. (See Figure 5-2.) The computer follows one of two prongs: the prong that prints `You win` or the prong that prints `You lose`. The computer decides which prong to take by testing the truth or falsehood of a *condition.* In Listing 5-1, the condition being tested is

```
inputNumber == randomNumber
```

Does the value of `inputNumber` equal the value of `randomNumber`? When the condition is true, the computer does the stuff between the condition and the word *else.* When the condition turns out to be false, the computer does the stuff after the word *else.* Either way, the computer goes on to execute the last `println` call, which displays `Thank you for playing`.

REMEMBER

The condition in an `if` statement must be enclosed in parentheses. However, a line like `if (inputNumber == randomNumber)` is not a complete statement (just as "If I had a hammer" isn't a complete sentence). So, this line `if (inputNumber == randomNumber)` shouldn't end with a semicolon.

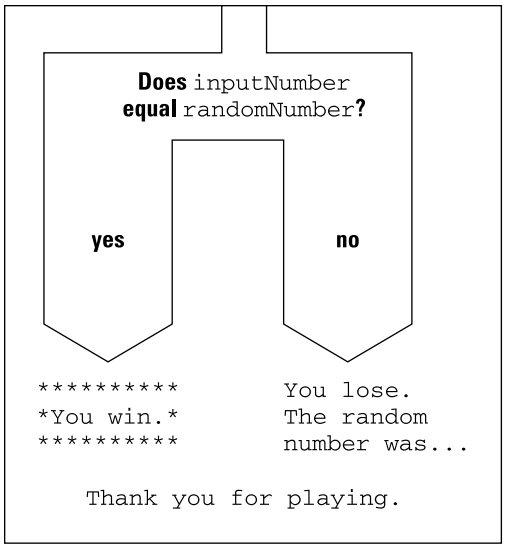


FIGURE 5-2: An `if` statement is like a fork in the road.

TECHNICAL STUFF

Sometimes, when I'm writing about a condition that's being tested, I slip into using the word *expression* instead of *condition.* That's okay because every condition is an expression. An expression is something that has a value and, sure enough, every condition has a value. The condition's value is either `true` or `false`. (For revealing information about expressions and values like `true` and `false`, see Chapter 4.)

Equal, equal

In Listing 5-1, in the `if` statement's condition, notice the use of the double equal sign. Comparing two numbers to see whether they're the same isn't the same as setting something equal to something else. That's why the symbol to compare for equality isn't the same as the symbol that's used in an assignment or an initialization. In an `if` statement's condition, you can't replace the double equal sign with a single equal sign. If you do, your program just won't work. (You almost always get an error message when you try to compile your code.)

On the other hand, if you never make the mistake of using a single equal sign in a condition, you're not normal. Several years ago, while I was teaching an introductory Java course, I promised that I'd swallow my laser pointer if no one made the single-equal-sign mistake during any of the lab sessions. This wasn't an idle promise. I knew I'd never have to keep it. As it turned out, even if I had ignored the first ten times anybody made the single-equal-sign mistake during those lab sessions, I would still be laser-pointer-free. Everybody mistakenly uses the single equal sign several times in a programming career.

TIP

The trick is not to avoid making the single-equal-sign mistake; the trick is to catch the mistake whenever you make it.

Brace yourself

The `if` statement in Listing 5-1 has two halves: a top half and a bottom half. I have names for these two parts of an `if` statement. I call them the *if part* (the top half) and the *else part* (the bottom half).

The `if` part in Listing 5-1 seems to have more than one statement in it. I make this happen by enclosing the three statements of the `if` part in a pair of curly braces. When I do this, I form a block. A *block* is a bunch of statements scrunched together by a pair of curly braces.

With this block, three calls to `println` are tucked away safely inside the `if` part. With the curly braces, the rows of asterisks and the words `You win` display only when the user's guess is correct.

This business with blocks and curly braces applies to the `else` part as well. In Listing 5-1, whenever `inputNumber` doesn't equal `randomNumber`, the computer executes three `print/println` calls. To convince the computer that all three of these calls are inside the `else` clause, I put these calls into a block. That is, I enclose these three calls in a pair of curly braces.

TECHNICAL STUFF

Strictly speaking, Listing 5-1 has only one statement between the `if` and `else` statements and only one statement after the `else` statement. The trick is that when you place a bunch of statements inside curly braces, you get a block; and a block behaves, in all respects, like a single statement. In fact, the official Java documentation lists blocks as one of the many kinds of statements. So, in Listing 5-1, the block that prints `You win` and asterisks is a single statement that has, within it, three smaller statements.

Your intent to indent

Notice how, in Listing 5-1, the `print` and `println` calls inside the `if` statement are indented. (This includes both the `You win` and `You lose` statements. The `print` and `println` calls that come after the word `else` are still part of the `if` statement.) Strictly speaking, you don't have to indent the statements that are inside an `if` statement. For all the compiler cares, you can write your whole program on a single line or place all your statements in an artful, misshapen zigzag. The problem is that neither you nor anyone else can make sense of your code if you don't indent your statements in some logical fashion. In Listing 5-1, the indenting of the `print` and `println` statements helps your eye (and brain) see quickly that these statements are subordinate to the overall `if/else` flow.

In a small program, unindented or poorly indented code is barely tolerable. But in a complicated program, indentation that doesn't follow a neat, logical pattern is a big, ugly nightmare.

REMEMBER

Many Java IDEs have tools to indent your code automatically. In your favorite IDE, look for menus such as Source➪ Format or Code➪ Reformat Code.

WARNING

When you write `if` statements, you may be tempted to chuck out the window all the rules about curly braces and simply rely on indentation. This strategy works in other programming languages, such as Python and Haskell, but not in Java. If you indent three statements after the word `else` and forget to enclose those statements in curly braces, the computer thinks that the `else` part includes only the first of the three statements. What's worse, the indentation misleads you into believing that the `else` part includes all three statements. This makes it more difficult for you to figure out why your code isn't behaving the way you think it should. Watch those braces!

Elseless in Helsinki

Okay, so the title of this section is contrived. Big deal! The idea is that you can create an `if` statement without the `else` part. Take, for instance, the code in Listing 5-1, shown earlier. Maybe you'd rather not rub it in whenever the user loses the game. The modified code in Listing 5-2 shows you how to do this (and Figure 5-3 shows you the result).

```
Pick a number from 1 to 10: 3
That was a very good guess 😄
The random number was 5.
Thank you for playing.


Pick a number from 1 to 10: 9
*You win.*
That was a very good guess 😄
The random number was 9.
Thank you for playing.
```

FIGURE 5-3: Two runs of the game in Listing 5-2.

LISTING 5-2: A Kinder, Gentler Guessing Game

```
void main() {
    String input = readln("Pick a number from 1 to 10: ");
    int inputNumber = Integer.parseInt(input);
    int randomNumber = new Random().nextInt(10) + 1;

    if (inputNumber == randomNumber) {
        println("*You win.*");
    }
    println("That was a very good guess \uD83D\uDE00");
    print("The random number was ");
    println(randomNumber + ".");

    println("Thank you for playing.");
}
```

The `if` statement in Listing 5-2 has no `else` part. When `inputNumber` is the same as `randomNumber`, the computer prints `You win`. When `inputNumber` is different from `randomNumber`, the computer doesn't print `You win`.

Notice the cryptic-looking code `\uD83D\uDE00` near the middle of Listing 5-3. That's Java's Unicode representation of the grinning-face emoji. You can see the result in Figure 5-3. As of March 2024, Java's Unicode implementation includes the 65,536 characters you can represent with 16 bits, plus an additional 84,277 characters known as *surrogates*. You can even use Unicode characters in your Java code's variable names. (See Figure 5-4.)

```
int איינס = 1;
int צווי = 2;
int דריי = איינס + צווי;
System.out.print("ענטפער : ");
System.out.println(דריי);
```

FIGURE 5-4: Right-to-left text meets left-to-right text.

TIP

You can see an official list of emojis by visiting `https://unicode.org/emoji/charts/full-emoji-list.html`.

ERRARE NECESSE EST UT RECTE FACIAS

TRY IT OUT

This experiment has three parts:

» Find out what happens when you run the code in Listing 5-1 and input the decimal representation of three-and-a-half (`3.5` or `3,5`, or something else, depending on your country's standard).

- Find out what happens when you run the code in Listing 5-1 and input a blank space followed by the digit `7`.
- Modify a line of code in Listing 5-1 as follows:

```
int inputNumber = Integer.parseInt(input.trim());
```

  Run the modified code and input a blank space followed by the digit `7`.

HOW DO YOU WRITE DECIMAL NUMBERS?

Run the following program:

```
void main() {
    String input = readln("What's the value of pi? ");
    double pi = Double.parseDouble(input);
    println("Thanks!");
}
```

Does it display `Thanks!` or give you an error message? If you get an error message, your country's way of representing decimal numbers may be different from the way I represent decimals in this book. Rather than type `3.14`, try typing `3,14` or some variation that you usually use.

STRAIGHT TALK

In Chapter 4, Listing 4-5 tells you whether you can or cannot fit ten people on an elevator. A run of the listing's code looks something like this:

```
True or False?
You can fit all ten of the
Brickenchicker decuplets
on the elevator:

false
```

Use what you know about Java's `if` statements to make the program's output more natural. Depending on the value of the program's `elevatorWeightLimit` variable, the output should be either

```
You can fit all ten of the
Brickenchicker decuplets
on the elevator.
```

or

```
You can't fit all ten of the
Brickenchicker decuplets
on the elevator.
```

Forming Conditions with Comparisons and Logical Operators

The Java programming language has plenty of little squiggles and doodads for your various condition-forming needs. This section tells you all about them.

Comparing numbers; comparing characters

Table 5-1 shows you the operators you can use to compare one value with another.

TABLE 5-1 Comparison Operators

Operator Symbol	Meaning	Example
`==`	is equal to	`numberOfCows == 5`
`!=`	is not equal to	`buttonClicked != panicButton`
`<`	is less than	`numberOfCows < 5`
`>`	is greater than	`myInitial > 'B'`
`<=`	is less than or equal to	`numberOfCows <= 5`
`>=`	is greater than or equal to	`myInitial >= 'B'`

You can use all of Java's comparison operators to compare numbers and characters. When you compare numbers, things go pretty much the way you think they should go. But when you compare characters, things are a little strange. Comparing uppercase letters with one another is no problem. Because the letter *B* comes alphabetically before *H*, the condition `'B' < 'H'` is true. Comparing lowercase letters with one another is also okay. What's strange is that when you compare an uppercase letter with a lowercase letter, the uppercase letter is always smaller. So, even though `'Z' < 'A'` is `false`, `'Z' < 'a'` is `true`.

TECHNICAL STUFF

Under the hood, the letters *A* through *Z* are stored with numeric codes 65 through 90. The letters *a* through *z* are stored with codes 97 through 122. That's why each uppercase letter is smaller than each lowercase letter.

WARNING

Be careful when you compare two numbers for equality (with `==`) or inequality (with `!=`). After you do some calculations and obtain two `double` values or two `float` values, the values that you have are seldom dead-on equal to one another. (The problem comes from those pesky digits beyond the decimal point.) For instance, the Fahrenheit equivalent of 21 degrees Celsius is 69.8, and when you calculate `9.0 / 5 * 21 + 32` by hand, you get 69.8. But the condition `9.0 / 5 * 21 + 32 == 69.8` turns out to be false. That's because, when the computer calculates `9.0 / 5 * 21 + 32`, it gets 69.80000000000001, not 69.8.

Comparing objects

When you start working with objects, you find that you can use `==` and `!=` to compare objects with one another. For instance, a button you see on the computer screen is an object. You can ask whether the thing that was just mouse-clicked is a particular button on your screen. You do this with Java's equality operator:

```
if (click.getSource() == submitButton) {
    Desktop.getDesktop().browse(new URI(
            "https://JavaForDummies.allmycode.com"));
}
```

CROSS REFERENCE

To find out more about responding to button clicks, read Chapter 16.

The big gotcha with Java's comparison scheme comes when you compare two strings. (For a word or two about Java's `String` type, see the section about reference types in Chapter 4.) When you compare two strings with one another, you don't want to use the double equal sign. Using the double equal sign would ask, "Is this string stored in exactly the same place in memory as that other string?" Usually, that's not what you want to ask. Instead, you usually want to ask, "Does this string have the same characters in it as that other string?" To ask the second question (the more appropriate question), Java's `String` type has a method named `equals`. (Like everything else in the known universe, this `equals` method is defined in the Java API.) The `equals` method compares two strings to see whether they have the same characters in them. For an example using Java's `equals` method, see Listing 5-3. (Figure 5-5 shows a run of the program in Listing 5-3.)

```
What's the password? swordfish
You typed >>swordfish<<

The word you typed is not
stored in the same place as
the real password, but that's
no big deal.

The word you typed has the
same characters as the real
password. You can use our
precious system.
```

FIGURE 5-5: The results of using == and using Java's `equals` method.

LISTING 5-3: Checking a Password

```
void main() {
    String password = readln("What's the password? ");

    println("You typed >>" + password + "<<");
    println("");

    if (password == "swordfish") {
        println("""
            The word you typed is stored in the same place
            as the real password. You must be a hacker.""");
    } else {
        println("""
            The word you typed is not stored in the same place
            as the real password, but that's no big deal.""");
    }
    println("");

    if (password.equals("swordfish")) {
        println("""
            The word you typed has the same characters as the
            real password. You can use our precious system.""");
    } else {
        println("""
            The word you typed doesn't have the same characters as
            the real password. You can't use our precious system.""");
    }
}
```

In Listing 5-3, the call `readln("What's the password? ")` grabs whatever word the user types on the computer keyboard. The code shoves the user's string input into the variable named *password.* Then the program's `if` statements use two different techniques to compare `password` with the string literal `"swordfish"`.

The more appropriate of the two techniques uses Java's `equals` method. The `equals` method looks funny because when you call it, you put a dot after one string and put the other string in parentheses. But that's the way you have to do it.

Wait! What about `Integer.parseInt`? I made a big fuss about that when I used it in Listing 5-1. Why don't you need `Integer.parseInt` here in Listing 5-3? To answer this question, look at the variable types in Listing 5-3. Since `readln` produces a `String` instance, the `password` variable has the same type as the `"swordfish"` literal. This is reinforced by the words `String password` near the start of Listing 5-3. There's no reason to convert anything to a numeric value in Listing 5-3. (In fact, calling `Integer.parseInt` to turn `password` into a number would be a very bad idea.)

In calling Java's `equals` method, it doesn't matter which string gets the dot and which gets the parentheses. For instance, in Listing 5-3, you could have written

```
if ("swordfish".equals(password))
```

The method would work just as well.

A call to Java's `equals` method looks imbalanced, but it's not. There's a reason behind the apparent imbalance between the dot and the parentheses. The idea is that you have two objects: the `password` object and the `"swordfish"` object. Each of these two objects is of type `String`. (However, `password` is a variable of type `String`, and `"swordfish"` is a `String` literal.) When you write `password.equals("swordfish")`, you're calling an `equals` method that belongs to the `password` object. When you call that method, you're feeding `"swordfish"` to the method as the method's parameter (pun intended). You can read more about methods belonging to objects in Chapter 7.

TIP

In addition to its `equals` method, Java has an `equalsIgnoreCase` method. Even though `"SWORDFISH".equals("swordfish")` is `false`, its close cousin `"SWORDFISH".equalsIgnoreCase("swordfish")` is `true`.

WARNING

The code in Listing 5-3 does a good job with `if` statements, but it does a terrible job with real-world user authentication. First, never show a password in plain view (without asterisks to masquerade the password). Second, don't handle passwords without encrypting them. Third, don't put passwords in the code. Fourth . . . well, I could go on and on. The code in Listing 5-3 just isn't meant to illustrate good username/password practices.

When one line isn't enough

A *text block* is a bunch of text surrounded on both sides by three double quotes (`"""`).

```
println("""
    The word you typed is stored in the same place
    as the real password. You must be a hacker.""");
```

A text block starts with three double quotation marks, and the remainder of that block's first line must be blank. If you mistakenly put nonblank text after those first three quotation marks, Java becomes sick to its stomach:

```
// Don't do this:
println("""The word you typed is stored in the same place
    as the real password. You must be a hacker.""");
```

This rule about having only blanks after the first three quotation marks helps with indentation issues. Take a look at the following code:

```
println("""
                    I. Controlling Program Flow
                       A. Introduction
                       B. Making Decisions
                          1. Guess the Number
                          2. Equal, equal
                       C. Forming Conditions""");
```

With the enormous 21-character indentation of the `I. Controlling Program Flow` line, the output of the code looks like this:

```
I. Controlling Program Flow
   A. Introduction
   B. Making Decisions
      1. Guess the Number
      2. Equal, equal
   C. Forming Conditions
```

Text blocks are useful because the text inside a block can straddle more than one line. Without text blocks, you may be tempted to put one quotation mark at each end, but that doesn't work. The following code, with traditional Java string notation, is forbidden:

```
// This code is incorrect:
println("
The word you typed is stored in the same place
    as the real password. You must be a hacker.");
```

This book's seventh edition hit the shelves in 2017, before Java had text blocks. In that edition, Listing 5-3 had a truckload of `println` calls:

```
println("The word you typed is stored in the same place");
println("as the real password. You must be a hacker.");
```

That was some ugly code!

Java's logical operators

Mr. Spock would be pleased: Java has all the operators you need for mixing and matching logical tests. The operators are shown in Table 5-2.

TABLE 5-2 Logical Operators

Operator Symbol	What It Means	Example
`&&`	and	`5 < x && x < 10`
`||`	or	`x < 5 || 10 < x`
`!`	not	`!password.equals("swordfish")`

You can use these operators to form all kinds of elaborate conditions. Listing 5-4 has an example.

LISTING 5-4: Checking Username and Password

```
import javax.swing.JOptionPane;

void main() {
    String username = JOptionPane.showInputDialog("Username:");
    String password = JOptionPane.showInputDialog("Password:");

    if (
           username != null && password != null
           &&
           (
             (username.equals("bburd") && password.equals("swordfish"))
             ||
             (username.equals("hritter") && password.equals("preakston"))
           )
       )
```

```
    {
        JOptionPane.showMessageDialog(null, "You're in.");
    }

    else {
        JOptionPane.showMessageDialog(null, "You're suspicious.");
    }
}
```

Several runs of the program in Listing 5-4 are shown in Figure 5-6. When the username is *bburd* and the password is *swordfish* or when the username is *hritter* and the password is *preakston*, the user sees a nice message. Otherwise, the user is a bum who sees the nasty message they deserve.

FIGURE 5-6: Several runs of the code from Listing 5-4.

Listing 5-4 illustrates a new way to get user input; namely, to show the user an input dialog box. The statement

```
String password = JOptionPane.showInputDialog("Password:");
```

in Listing 5-4 performs more or less the same task as the statement

```
String password = readln("What's the password? ");
```

from Listing 5-3. The big difference is, while `readln` displays dull-looking text in a console, `JOptionPane.showInputDialog` displays a fancy dialog box containing a text field and buttons. (Compare Figures 5-5 and 5-6.) When the user clicks OK, the computer takes whatever text is in the text field and hands that text over to a variable. In fact, Listing 5-4 uses `JOptionPane.showInputDialog` twice — once to get a value for the username variable and a second time to get a value for the password variable.

Near the end of Listing 5-4, I use a slight variation on the `JOptionPane` business:

```
JOptionPane.showMessageDialog(null, "You're in.");
```

With `showMessageDialog`, I show a simple dialog box — a box with no text field. (Again, see Figure 5-6.)

TIP

If you want to look like an experienced Java programmer, don't imitate the code in Listing 5-4. For one thing, the indentation and line breaks in Listing 5-4 are unusual. I wrote the `if` statement this way so that you can easily see how the statement's parts relate to one another, but professional Java programmers would balk at this formatting. Aside from the strange indentation, Listing 5-4 uses the outdated `JOptionPane` class to display a window on the computer screen. Modern Java programs shun `JOptionPane` and the other `javax.swing` classes in favor of a platform called *JavaFX*. I used `JOptionPane` because it's good for small, single-window programs like the one in Listing 5-4. But JavaFX is much better for elaborate desktop designs. For the skinny on JavaFX, see Chapter 16.

Vive les nuls!

The French translations of *For Dummies* books are books *Pour les Nuls.* So a "dummy" in English is a "nul" in French.* But in Java, the word `null` means "nothing." When you see

```
if (
        username != null
```

in Listing 5-4, you can imagine that you see

```
if (
        username isn't nothing
```

or

```
if (
        username has any value at all
```

To find out how usernames can have no value, see the last row in Figure 5-6. When you click Cancel in the first dialog box, the computer hands `null` to your program. So, in Listing 5-4, the variable username becomes `null`. The comparison `username != null` checks to make sure you haven't clicked Cancel in the program's first dialog box. The comparison `password != null` performs the same kind of check for the program's second dialog box. When you see the `if` statement in Listing 5-4, you can imagine that you see the conditions in Figure 5-7.

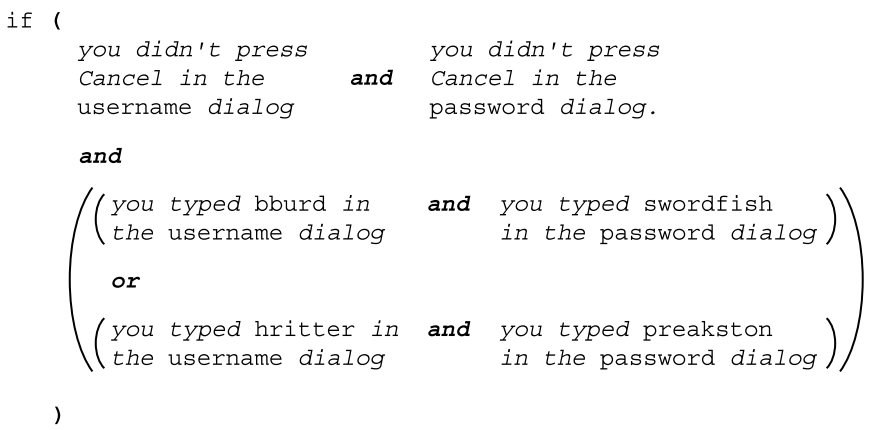

if (
you didn't press Cancel in the username dialog
and
you didn't press Cancel in the password dialog.
and
you typed bburd in the username dialog
and
you typed swordfish in the password dialog
or
you typed hritter in the username dialog
and
you typed preakston in the password dialog
)

FIGURE 5-7: Interpreting the code in Listing 5-4.

WARNING

In Listing 5-4, the comparisons `username != null` and `password != null` are not optional. If you forget to include these and then click Cancel when the program runs, you get a nasty `NullPointerException` message, and the program comes crashing down before your eyes. The word `null` represents nothing, and in Java, you can't compare nothing to a string like `"bburd"` or `"swordfish"`. In Listing 5-4, the purpose of the comparison `username != null` is to prevent Java from moving on to check `username.equals("bburd")` whenever you happen to click Cancel. Without this preliminary `username != null` test, you're courting trouble.

---

* In Russian, a "dummy" is a "чайник," which, when interpreted literally, means a "teapot." So, in Russian, this book is *Java For Teapots.* I've never been called a teapot, and I'm not sure how I'd react if I were.

TECHNICAL STUFF

The last couple of `null`s in Listing 5-4 are different from the others. In the code `JOptionPane.showMessageDialog (null, "You're in.")`, the word `null` stands for "no other dialog box." In particular, the call `showMessageDialog` tells Java to pop up a new dialog box, and the word `null` indicates that the new dialog box doesn't grow out of any existing dialog box. One way or another, Java insists that you say something about the origin of the newly popped dialog box. (For some reason, Java doesn't insist that you specify the origin of the `showInputDialog` box. Go figure!) Anyway, in Listing 5-4, having a `showMessageDialog` box pop up from nowhere is quite useful.

(Conditions in parentheses)

Keep an eye on those parentheses! When you're combining conditions with logical operators, it's better to waste typing effort and add unneeded parentheses than to goof up your result by using too few parentheses. Take, for example, this expression:

```
2 < 5 || 100 < 6 && 27 < 1
```

By misreading this expression, you might conclude that the expression is false. That is, you could wrongly read the expression as meaning `(`*`something-or-other`*`) && 27 < 1`. Because `27 < 1` is false, you would conclude that the whole expression is false. The fact is that, in Java, any `&&` operator is evaluated before any `||` operator. So the expression really asks whether `2 < 5 || (`*`something-or-other`*`)`. Because `2 < 5` is true, the whole expression is true.

To change the expression's value from `true` to `false`, you can put the expression's first two comparisons in parentheses, like this:

```
(2 < 5 || 100 < 6) && 27 < 1
```

TIP

Java's `||` operator is *inclusive.* This means that you get a `true` value whenever the thing on the left side is true, the thing on the right side is true, or both things are true. For instance, the expression `2 < 10 || 20 < 30` is true.

WARNING

In Java, you can't combine comparisons the way you do in ordinary English. In English, you may say, "We'll have between three and ten people at the dinner table." But in Java, you get an error message if you write `3 <= people <= 10`. To do this comparison, you need something like `3 <= people && people <= 10`.

In Listing 5-4, the `if` statement's condition has more than a dozen parentheses. What happens if you omit two of them?

```
if (
        username != null && password != null &&
        // open parenthesis omitted
            (username.equals("bburd") && password.equals("swordfish")) ||
            (username.equals("hritter") && password.equals("preakston"))
        // close parenthesis omitted
)
```

Java tries to interpret your wishes by grouping everything before the "or" (the `||` operator):

```
if (
        username != null && password != null &&
        (username.equals("bburd") && password.equals("swordfish"))

        ||

        (username.equals("hritter") && password.equals("preakston"))
   )
```

When the user clicks Cancel and `username` is `null`, Java says, "Okay! The stuff before the `||` operator is false, but maybe the stuff after the `||` operator is true. I'll check the stuff after the `||` operator to find out whether it's true." (Java often talks to itself. The psychiatrists are monitoring this situation.)

Anyway, when Java finally checks `username.equals("hritter")`, your program aborts with an ugly `NullPointerException` message. You've made Java angry by trying to apply `.equals` to a `null` username. (Psychiatrists have recommended anger management sessions for Java, but Java's insurance plan refuses to pay for the sessions.)

An import declaration

It's always good to announce your intentions upfront. Consider the following classroom lecture:

> *Today, in our History of Film course, we'll discuss the career of actor* ***Lionel Herbert Blythe Barrymore****.*

> *Born in Philadelphia,* ***Barrymore*** *appeared in more than 200 films, including* It's a Wonderful Life, Key Largo, and Dr. Kildare's Wedding Day. *In addition,* ***Barrymore*** *was a writer, composer, and director. Barrymore did the voice of Ebenezer Scrooge every year on radio. . . .*

Interesting stuff, heh? Now compare these paragraphs with a lecture in which the instructor doesn't begin by introducing the subject:

> *Welcome once again to the History of Film.*
>
> *Born in Philadelphia,* ***Lionel Barrymore*** *appeared in more than 200 films, including* It's a Wonderful Life, Key Largo, *and* Dr. Kildare's Wedding Day. *In addition,* ***Barrymore (not Ethel, John, or Drew)*** *was a writer, composer, and director.* ***Lionel Barrymore*** *did the voice of Ebenezer Scrooge every year on radio. . . .*

Without a proper introduction, a speaker may have to remind you repeatedly that the discussion is about Lionel Barrymore and not about any other Barrymore. The same is true in a Java program. Look again at the beginning of Listing 5-4:

```
import javax.swing.JOptionPane;

void main() {
    String username = JOptionPane.showInputDialog("Username:");
    String password = JOptionPane.showInputDialog("Password:");
```

The first line of Listing 5-4 is an *import declaration*. The import declaration announces that you'll be using `JOptionPane` in this Java program. The declaration clarifies what you mean by `JOptionPane` with the class's full name `javax.swing.JOptionPane`. (Hey! Didn't the first lecturer clarify with the full name "Lionel Herbert Blythe Barrymore"?) After announcing your intentions in the import declaration, you can use the abbreviated name `JOptionPane` in the remainder of Listing 5-4.

Like thousands of other names, the name `javax.swing.JOptionPane` is defined in Java's API. So, to use the name `JOptionPane` throughout Listing 5-4, I import `javax.swing.JOptionPane` at the top of the listing. If you don't use this import declaration, you have to repeat the full `javax.swing.JOptionPane` name wherever you use the name `JOptionPane` in your code. For example, without an import declaration, the start of Listing 5-4 would look like this:

```
void main() {
    String username = javax.swing.JOptionPane.showInputDialog("Username:");
    String password = javax.swing.JOptionPane.showInputDialog("Password:");
```

TECHNICAL STUFF

To be precise, `javax.swing.JOptionPane` is the *fully qualified name* of a class whose *simple name* is `JOptionPane`.

Every class in Java's API has a fully qualified name. For example, the fully qualified name of the familiar `String` class is `java.lang.String`. The `Random` class's close friends call it by its simple name `Random`. But the `Random` class's birth certificate has the class's fully qualified name — `java.util.Random`.

So your next question is, "Why do I have to import `javax.swing.JOptionPane` and not `java.lang.String` or `java.util.Random`? The answer is that some classes in Java's API creep into your code without the need for explicit import declarations. Java's API has thousands of classes, and each class belongs to a group of classes known as a *package*. Going one step further, each package belongs to a group of packages known as a *module*. Figure 5-8 shows you a few of the classes in the Java API.

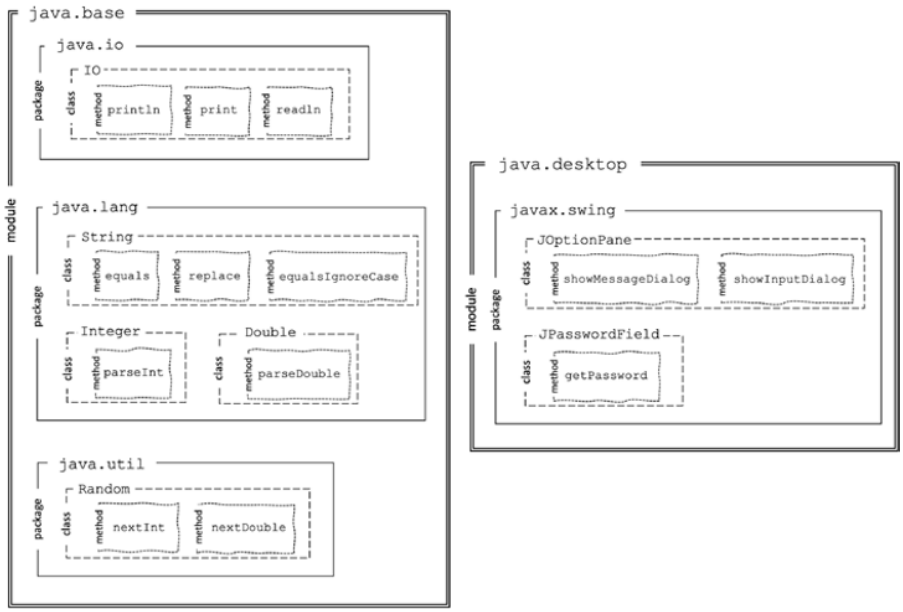


FIGURE 5-8: What-lives-where in Java's API.

Notice the box labeled `java.base` in Figure 5-8. With each example in Chapters 1–6 of this book, you don't need an import declaration for any class inside `java.base`. Java automatically imports all these classes for you. So, you don't start any of this chapter's examples by importing `String` or `Random`. But you start Listing 5-4 by importing `javax.swing.JOptionPane` because `JOptionPane` isn't inside `java.base`.

Warning! Explicit content!

Here's an official explanation of the `import` situation in Listing 5-4: A Java program has explicit elements and implicit elements.

- **An *explicit* element corresponds to some text in the program.**

  At the start of Listing 5-4, the line `import javax.swing.JOptionPane` is an explicit import declaration. For that matter, each part of the code in Listing 5-4 stands for an explicit something-or-other.

- **An *implicit* element is something that Java adds behind the scenes, without corresponding to any text in the program.**

  Even though the line `import java.lang.String` doesn't appear in Listing 5-4, Java behaves as if the listing contains that line. This comes from the rule that Java implicitly declares everything in `java.base` for programs like the one in Listing 5-4.

Starting in Chapter 7, you can find listings that don't implicitly import all the classes inside the `java.base` module. Instead, these listings implicitly import only classes inside the `java.lang` package. (Refer to Figure 5-8.) So, in some of these listings, you don't need an import declaration for `String`, but you do need import declarations for `Random`. Sorry about that. All good things must come to an end!

There's another wrinkle in this story about import declarations. It involves `println`, `Integer`, `Double`, and many other names. My advice is for you to ignore the wrinkle until you read about `static` in Chapter 11.

REMEMBER

Understanding import declarations isn't easy. But fortunately, many IDEs have convenient helper features for import declarations. In your IDE of choice, look for menus such as Code➪Optimize Imports or Source➪Organize Imports.

CROSS REFERENCE

For more details about the use of import declarations, see Chapters 7, 11, and 12.

Make some changes to the code in Listing 5-4.

TRY IT OUT

THE RULE OF THREE

Add a third username/password combination to the list of acceptable logins.

OUT, DAMN'D NOT!

In Listing 5-4, change

```
username != null && password != null
```

to

```
!(username == null || password == null)
```

Does the program still work? Why or why not?

EQUAL BYTES

In Listing 5-4, change

```
username != null && password != null
```

to

```
!(username == null && password == null)
```

This is almost the same as the previous experiment. The only difference is the use of `&&` instead of `||` between the two `== null` tests. Does the program still work? Why or why not?

PASSWORDS IN PLAIN SIGHT

Run the following code and find out what it does:

```
import javax.swing.JOptionPane;
import javax.swing.JPasswordField;

void main() {
    JPasswordField passwordField = new JPasswordField();
    JOptionPane.showMessageDialog(null,
            new Object[] {"Password", passwordField});
    String userInput = new String(passwordField.getPassword());
    if (userInput.equals("swordfish")) {
        JOptionPane.showMessageDialog(null, "You're in.");
    } else {
        JOptionPane.showMessageDialog(null, "You're suspicious.");
    }
}
```

The Nesting Habits of if Statements

Have you seen those cute Russian matryoshka nesting dolls? Open one, and another one is inside. Open the second, and a third one is inside it. You can do the same thing with Java's `if` statements. (Talk about fun!) Listing 5-5 shows you how.

LISTING 5-5: Nested if Statements

```
void main() {
    String username = readln("Username: ");
    if (username.equals("bburd")) {
        String password = readln("Password: ");
        if (password.equals("swordfish")) {
            println("You're in.");
        } else {
            println("Incorrect password");
        }
    } else {
        println("Unknown user");
    }
}
```

Figure 5-9 shows several runs of the code in Listing 5-5. The main idea is that to log on, you have to pass two tests. (In other words, two conditions must be true.) The first condition tests for a valid username; the second condition tests for the correct password. If you pass the first test (the username test), you march right into another `if` statement that performs a second test (the password test). If you fail the first test, you never make it to the second test. Figure 5-10 shows the overall plan.

```
Username: bburd
Password: swordfish
You're in.


Username: bburd
Password: catfish
Incorrect password


Username: jschmoe
Unknown user
```

FIGURE 5-9: Three runs of the code in Listing 5-5.

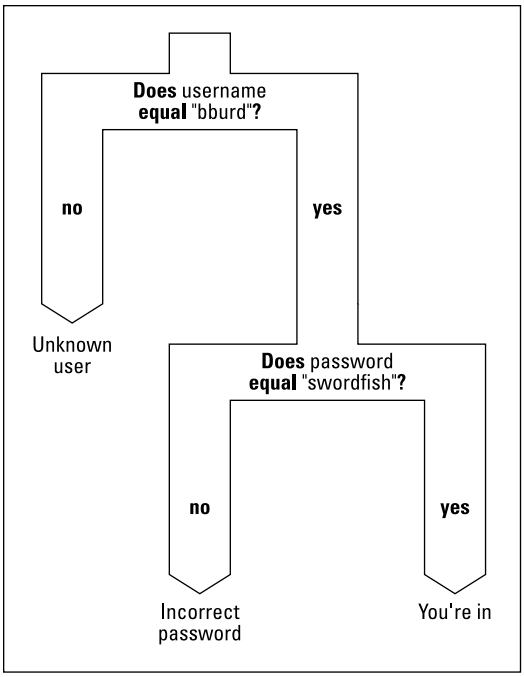


FIGURE 5-10: Don't eat with this fork.

I CHANGED MY MIND

TRY IT OUT

Modify the program in Listing 5-4 so that if the user clicks Cancel for either the username or the password, the program replies with a `Not enough information` message.

Choosing among Many Alternatives

I'm the first to admit that I hate making decisions. If things go wrong, I would rather have the problem be someone else's fault. Writing the previous sections (on making decisions with Java's `if` statement) knocked the stuffing right out of me. That's why my mind boggles as I begin this section on choosing among many alternatives. What a relief it is to have that confession out of the way!

Java's glorious switch statement

Now it's time to explore situations in which you have a decision with many branches. Take, for instance, the popular campfire song "Al's All Wet." (For a

review of the lyrics, see the nearby "Al's All Wet" sidebar.) You're eager to write code that prints this song's lyrics. Fortunately, you don't have to type all the words over and over again. Instead, you can take advantage of the repetition in the lyrics.

"AL'S ALL WET"

Sung to the tune of "Gentille Alouette"

Al's all wet. Oh, why is Al all wet? Oh,
Al's all wet 'cause he's standing in the rain.
Why is Al out in the rain?
That's because he has no brain.
Has no brain, has no brain,
In the rain, in the rain.

Oh, oh, oh, oh

Al's all wet. Oh, why is Al all wet? Oh,
Al's all wet 'cause he's standing in the rain.
Why is Al out in the rain?
That's because he is a pain.
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.

Oh, oh, oh, oh

Al's all wet. Oh, why is Al all wet? Oh,
Al's all wet 'cause he's standing in the rain.
Why is Al out in the rain?
'Cause this is the last refrain.
Last refrain, last refrain,
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.

Oh, oh, oh, oh

Al's all wet. Oh, why is Al all wet? Oh,
Al's all wet 'cause he's standing in the rain.

—Harriet Ritter and Barry Burd

To display parts of the song's lyrics, you create a variable named `verse`. The value of `verse` is `1`, `2`, or `3`, depending on which verse of "Al's All Wet" you're trying to print. You could have a big, clumsy bunch of `if` statements that checks each possible `verse` number:

```
if (verse == 1) {
    println("That's because he has no brain.");
}
if (verse == 2) {
    println("That's because he is a pain.");
}
if (verse == 3) {
    println("'Cause this is the last refrain.");
}
```

But that approach seems wasteful. Why not create a statement that checks the value of `verse` just once and then takes an action based on the value it finds? Fortunately, just such a statement exists — it's called a *switch* statement. Listing 5-6 has an example of a `switch` statement.

LISTING 5-6: **A switch Statement**

```
void main() {
    int verse = Integer.parseInt(readln("Which verse? "));
    switch (verse) {
        case 1 -> println("That's because he has no brain.");
        case 2 -> println("That's because he is a pain.");
        case 3 -> println("'Cause this is the last refrain.");
        default -> println("No such verse. Please try again.");
    }
    println("Oh, oh, oh, oh");
}
```

WARNING

Before Oracle released Java 14 in 2020, most of the features I describe in this section weren't available. If you stumble upon a reason to use an older version of Java, check the later section titled "Your grandparents' `switch` statement."

Figure 5-11 shows two runs of the program in Listing 5-6. (Figure 5-12 illustrates the program's overall idea.) First, the user types a number, like the number 2. Then execution of the program reaches the top of the `switch` statement. The computer checks the value of the `verse` variable. When the computer determines that the `verse` variable's value is `2`, the computer checks each case of the `switch` statement. The value `2` doesn't match the topmost case, so the computer proceeds to the middle of the three cases. The value posted for the middle case (the number 2)

matches the value of the `verse` variable, so the computer executes the statement in case 2:

```
println("That's because he is a pain.");
```

```
Which verse? 2
That's because he is a pain.
Oh, oh, oh, oh

Which verse? 6
No such verse. Please try again.
Oh, oh, oh, oh
```

FIGURE 5-11: Running the code from Listing 5-6 two times.

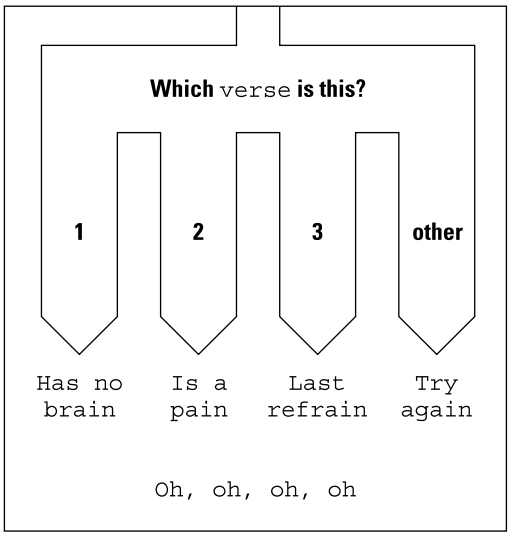


FIGURE 5-12: The big fork in the code of Listing 5-6.

If the pesky user asks for verse 6, the computer bypasses cases 1, 2, and 3. The computer goes straight to the default. In the default, the computer displays `No such verse. Please try again` and then jumps out of the `switch` statement. After the computer is out of the `switch` statement, the computer displays `Oh, oh, oh, oh`.

TIP

You can put more than one value in each `case` clause of a `switch` statement. You can also use curly braces to create a block of statements inside a `case` clause. Here's some sample code:

```
void main() {
    String reply = readln("Will you pay me? ");
```

```
    switch (reply.toLowerCase()) {
        case "yes", "y", "ok" -> {
            println("Thank you!");
            println("You're very nice.");
        }
        case "Maybe" -> println("Wha' da' ya' mean by that?");
        case "no", "n" -> {
            println("You're a deadbeat.");
            println("Thanks for nothing!");
        }
    }
}
```

This code responds correctly to `yes`, `y`, and `ok`. In fact, if the user types `YES` or even `yEs`, the value of `reply.toLowerCase()` is the lowercase string `"yes"`, so the program responds with `"Thank you!"` and `"You're very nice."`

Express yourself

If you look again at Listing 5-6, you wonder why someone took so much delight in typing the word `println`. That word appears four times in the `switch` statement, and the only difference is the choice of a sarcastic verse. This repeated use of `println` seems wasteful. Why not have only one call to `println` for all the different answers the program can display?

Java's `switch` expression addresses this issue. In fact, runs of the code in Listing 5-7 look exactly the same as the runs of Listing 5-6. (Refer to Figure 5-11.)

LISTING 5-7: Out with the Old println!

```
void main() {
    int verse = Integer.parseInt(readln("Which verse? "));

    String line;
    line = switch (verse) {
        case 1 -> "That's because he has no brain.";
        case 2 -> "That's because he is a pain.";
        case 3 -> "'Cause this is the last refrain.";
        default -> "No such verse. Please try again.";
    };

    println(line);
    println("Oh, oh, oh, oh");
}
```

In any computer language, the word *expression* refers to a bunch of code that stands for a value. In Listing 5-7, the entire `switch` expression stands for a string of characters. For example, when `verse` is `2`, the code

```
line = switch (verse) {
    case 1 -> "That's because he has no brain.";
    case 2 -> "That's because he is a pain.";
    case 3 -> "'Cause this is the last refrain.";
    default -> "No such verse. Please try again.";
};
```

stands for the string `"That's because he is a pain."` So, in that scenario, Listing 5-7 does the same thing as this code:

```
line = "That's because he is a pain.";
out.println(line);
```

In this small snippet of code, notice how the first line ends with a semicolon. That's why there's a semicolon at the end of the big `switch` expression in Listing 5-7.

You can squeeze even more goodness out of Java's `switch` expression. For example, you don't have to assign the expression's value to a variable. You can bypass the variable and print the `switch` expression's value:

```
println(switch (verse) {
    case 1 -> "That's because he has no brain.";
    case 2 -> "That's because he is a pain.";
    case 3 -> "'Cause this is the last refrain.";
    default -> "No such verse. Please try again.";
});
```

But remember that a `switch` expression can't stand independently on its own. The following code makes no sense:

```
// Bad code!
int verse = Integer.parseInt(readln("Which verse? "));
String line;

"That's because he is a pain.";   // This is not a complete statement.

println(line);
```

The code with a freestanding `switch` expression doesn't work, either.

```
// Bad code!
int verse = Integer.parseInt(readln("Which verse? "));
String line;

switch (verse) {                                    // This
    case 1 -> "That's because he has no brain.";    //   is
    case 2 -> "That's because he is a pain.";       //   not
    case 3 -> "'Cause this is the last refrain.";   //   a
    default -> "No such verse. Please try again."; //   complete
};                                                  //   statement.

println(line);
```

Your grandparents' switch statement

In versions of Java before Java 14, the `switch` constructs in Listings 5-6 and 5-7 wouldn't have worked. That's because Java's `switch` statement came originally from the `switch` statement in the C/C++ language family, and the C/C++ `switch` statement was quite clunky. The code in Listing 5-8 does the same thing as the code in Listings 5-6 and 5-7, but Listing 5-8 uses Java's old-style `switch` statement.

LISTING 5-8: **A la recherche du temps perdu**

```
void main() {
    int verse = Integer.parseInt(readln("Which verse? "));
    switch (verse) {
        case 1:
            println("That's because he has no brain.");
            break;
        case 2:
            println("That's because he is a pain.");
            break;
        case 3:
            println("'Cause this is the last refrain.");
            break;
        default:
            println("No such verse. Please try again.");
            break;
    }
    println("Oh, oh, oh, oh");
}
```

Runs of the code in Listing 5-8 look exactly the same as runs of Listings 5-6 and 5-7. (Refer to Figure 5-11.) The big difference is that Listing 5-8 uses the old form of the `switch` statement. In the old form:

- You use colons (`:`) instead of arrows (`->`).
- You need `break` statements to bypass any remaining cases.

When Java encounters a `break` statement, it jumps out of whatever `switch` statement it's in. For example, imagine that `verse` has the value `2`. In Listing 5-8, Java prints `That's because he is a pain`. Then, because of the `break` statement, Java skips right past the case that would display `'Cause this is the last refrain`. In fact, Java jumps out of the entire `switch` statement and goes directly to the statement just after the end of the `switch` statement. The computer displays `Oh, oh, oh, oh` because that's what the statement after the `switch` statement tells Java to do.

TIP

You don't really need to put a `break` at the very end of an old `switch` statement. In Listing 5-8, the last `break` (the `break` that's part of the default) is just for the sake of overall tidiness.

The need for `break` inside the old `switch` statement is a Java programmer's nightmare. Everyone forgets to add these `break` statements, and when you forget to add `break` statements, you get fall-through. With *fall-through*, execution of the code falls right through from one case to the next. Execution keeps falling through until you eventually reach a `break` statement or the end of the entire `switch` statement.

TIP

With `switch` statements from the olden days, you can sandwich as many statements as you want between two `case` clauses. For example, the following code snippet is perfectly legal:

```
case 1:
    print("That's because ");
    println("he has no brain.");
    println("... and because I don't like him.");
    break;
case 2:
```

Free fall

Usually, when you're using a `switch` statement, you don't want fall-through, so you pepper `break` statements throughout the `switch` statements. But,

occasionally, fall-through is just the thing you need. Take, for instance, the "Al's All Wet" song. (The classy lyrics are shown in the sidebar bearing the song's name.) Each verse of "Al's All Wet" adds new lines in addition to the lines from previous verses. This situation (accumulating lines from one verse to another) cries out for a `switch` statement with fall-through. Listing 5-9 demonstrates the idea.

LISTING 5-9: **A switch Statement with Fall-Through**

```
void main() {
    String verse = readln("Which verse? (one, two or three) ");
    switch (verse) {
        case "three": println("Last refrain, last refrain,");
        case "two":   println("He's a pain, he's a pain,");
        case "one":   println("Has no brain, has no brain,");
    }
    println("""
        In the rain, in the rain.
        Oh, oh, oh, oh""");
    println("");
}
```

Figure 5-13 shows several runs of the program in Listing 5-9. Because the `switch` has no `break` statements in it, fall-through happens all over the place. For instance, when the user enters the word `two`, Java starts by executing the statement in `case "two"`:

```
println("He's a pain, he's a pain,");
```

Then Java marches right on to execute the statement in `case "one"`:

```
println("Has no brain, has no brain,");
```

That's good because the song's second verse has all these lines in it.

Notice what happens when the user asks for verse six. The `switch` statement in Listing 5-9 has no `case "six"` and no `default`, so none of the actions inside the `switch` statement is executed. Even so, with statements that print `In the rain, in the rain` and `Oh, oh, oh, oh` right after the `switch` statement, the computer displays something when the user asks for verse `"six"`.

TRY IT OUT

Get some practice with `if` statements and `switch` statements!

```
Which verse? (one, two or three) one
Has no brain, has no brain,
In the rain, in the rain.
Oh, oh, oh, oh

Which verse? (one, two or three) two
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.
Oh, oh, oh, oh

Which verse? (one, two or three) three
Last refrain, last refrain,
He's a pain, he's a pain,
Has no brain, has no brain,
In the rain, in the rain.
Oh, oh, oh, oh

Which verse? (one, two or three) six
In the rain, in the rain.
Oh, oh, oh, oh
```

FIGURE 5-13: Running the code of Listing 5-9 four times.

MONTH-TO-MONTH RESUSCITATION

Write a program that inputs the name of a month and outputs the number of days in that month. In this first version of the program, assume that February always has 28 days. Use Java's `switch` statement — the kind I used in Listing 5-6.

LEAPING TO CONCLUSIONS

Make changes to your code from the earlier section "Month-to-Month Resuscitation." Have the user input a month name and also `yes` or `no` in response to the question `Is it a leap year?`

HOW ABOUT A DATE?

Make changes to your code from the earlier section "Leaping to Conclusions." Rather than mimic Listing 5-6, use a `switch` expression of the kind you find in Listing 5-7. Then use Java's old-style `switch` statement, like the one you see in Listing 5-8.

THE YIELD APP

Take this code for a spin and see what it does:

```
void main() {
    String month = readln("Which month? ");
    println(switch (month) {
        case "January", "March", "May", "July",
                "August", "October", "December" -> 31;
        case "April", "June", "September", "November" -> 30;
        case "February" -> {
            boolean isLeapYear =
                      Boolean.parseBoolean(readln("Leap year (true/false)? "));
            if (isLeapYear) {
                yield 29;
            } else {
                yield 28;
            }
        }
        default -> 0;
    } + " days");
}
```

IN THIS CHAPTER

- Using basic looping
- Counting as you loop
- Repeating relentlessly (until the user provides a clear answer)

Chapter 6
Controlling Program Flow with Loops

In 1966, the company that brings you Head & Shoulders shampoo made history. On the back of the bottle, the directions for using the shampoo read, "LATHER–RINSE–REPEAT." Never before had a complete set of directions (for doing anything, let alone shampooing your hair) been summarized so succinctly. People in the directions-writing business hailed this as a monumental achievement. Directions like these stood in stark contrast to others of the time. (For instance, the first sentence on a can of bug spray read, "Turn this can so that it points away from your face." Duh!)

Aside from their brevity, the thing that made the Head & Shoulders directions so cool was that, with three simple words, company leaders managed to capture a notion that's at the heart of all instruction-giving: repetition. That last word, *REPEAT,* took an otherwise bland instructional drone and turned it into a sophisticated recipe for action.

The fundamental idea is that when you're following directions, you don't just follow one instruction after another. Instead, you take turns in the road. You make decisions ("If HAIR IS DRY, then USE CONDITIONER") and you go into loops ("LATHER–RINSE, and then LATHER–RINSE again."). In computer programming, you use decision-making and looping all the time. This chapter explores looping in Java.

Repeating Instructions Over and Over Again (Java while Statements)

Here's a guessing game for you. The computer generates a random number from 1 to 10. The computer asks you to guess the number. If you guess incorrectly, the game continues. As soon as you guess correctly, the game is over. Listing 6-1 shows the program to play the game, and Figure 6-1 shows a round of play.

```
       ************
Welcome to the Guessing Game
       ************

Pick a number from 1 to 10: 2

Try again...
Pick a number from 1 to 10: 5

Try again...
Pick a number from 1 to 10: 8

Try again...
Pick a number from 1 to 10: 3
You win after 4 guesses.
```

FIGURE 6-1: Play until you drop.

LISTING 6-1: A Repeating Guessing Game

```
void main() {

    int numGuesses = 0;
    int randomNumber = new Random().nextInt(10) + 1;

    println("""
              ************
        Welcome to the Guessing Game
              ************""");
    println("");  // or println(); (See Chapter 4.)

    String input = readln("Pick a number from 1 to 10: ");
    int inputNumber = Integer.parseInt(input);
    numGuesses++;
```

```
    while (inputNumber != randomNumber) {
        println("");
        println("Try again...");
        input = readln("Pick a number from 1 to 10: ");
        inputNumber = Integer.parseInt(input);
        numGuesses++;
    }
    print("You win after ");
    println(numGuesses + " guesses.");
}
```

In Figure 6-1, the user makes four guesses. Each time around, the computer checks to see whether the guess is correct. An incorrect guess generates a request to try again. For a correct guess, the user gets a rousing `You win`, along with a tally of the number of guesses they made. The computer repeats several statements, checking each time through to see whether the user's guess is the same as a certain randomly generated number. Each time the user makes a guess, the computer adds 1 to its tally of guesses. When the user makes the correct guess, the computer displays that tally. Figure 6-2 illustrates the flow of action.

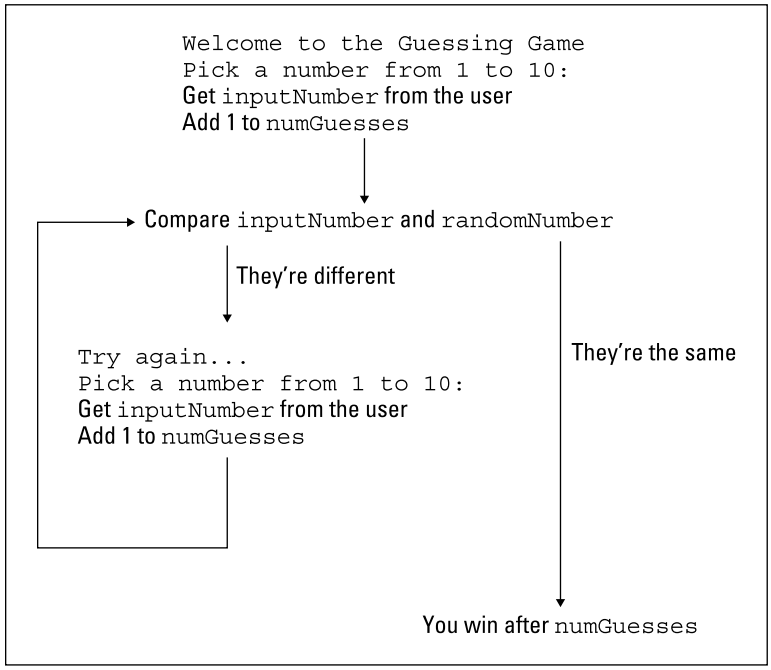

Welcome to the Guessing Game
Pick a number from 1 to 10:
Get inputNumber from the user
Add 1 to numGuesses
Compare inputNumber and randomNumber
They're different
Try again...
Pick a number from 1 to 10:
Get inputNumber from the user
Add 1 to numGuesses
They're the same
You win after numGuesses

FIGURE 6-2: Around and around you go.

When you look over Listing 6-1, you see the code that does all this work. At the core of the code is a thing called a *while statement* (also known as a *while loop*). Rephrased in English, the `while` statement says:

```
while inputNumber doesn't equal randomNumber, repeat what's in these braces {

}
```

The stuff in curly braces (the stuff that repeats) is the code that prints `Try again` and `Pick a number...`, gets a value from the keyboard, and adds 1 to the count of the user's guesses.

TIP

When you're dealing with counters, like `numGuesses` in Listing 6-1, you may easily become confused and be off by 1 in either direction. You can avoid this headache by making sure that the `++` statements stay close to the statements whose events you're counting. For example, in Listing 6-1, the variable `numGuesses` starts with a value of `0`. That's because, when the program starts running, the user hasn't made any guesses. Later in the program, right after each call to `readln`, is a `numGuesses++` statement. That's how you do it — you increment the counter as soon as the user enters another guess.

The statements in curly braces are repeated as long as `inputNumber != randomNumber` is true. Each repetition of the statements in the loop is called an *iteration* of the loop. In Figure 6-1, the loop undergoes three iterations. (If you don't believe that Figure 6-1 has exactly three iterations, count the number of `Try again` printings in the program's output. A `Try again` appears for each incorrect guess.)

When, at long last, the user enters the correct guess, the computer goes back to the top of the `while` statement, checks the condition in parentheses, and finds itself in double double-negative land. The not equal (`!=`) relationship between `inputNumber` and `randomNumber` no longer holds. In other words, the `while` statement's condition, `inputNumber != randomNumber`, is false. Because the `while` statement's condition is false, the computer jumps past the `while` loop and goes on to the statements just below the `while` loop. In these two statements, the computer prints `You win` after four guesses.

REMEMBER

With code of the kind shown in Listing 6-1, the computer never jumps out in mid-loop. When the computer finds that `inputNumber` isn't equal to `randomNumber`, the computer marches on and executes all five statements inside the loop's curly braces. The computer performs the test again (to see whether `inputNumber` is still not equal to `randomNumber`) only after it fully executes all five statements in the loop.

TRY IT OUT

I have two things for you to try:

BIGGER AND BETTER

Modify the program in Listing 6-1 so that the randomly generated number is a number from 1 through 100. To make life bearable for the game player, have the program display a hint whenever the player guesses incorrectly. Hints such as `Try a higher number` or `Try a lower number` are helpful.

NO NEGATIVITY

Write a program in which the user types `int` values, one after another. The program stops looping when the user types a number that isn't positive (for example, the number 0 or the number –17). After all the looping, the program displays the largest number that the user typed. For example, if the user types the numbers

```
7
25
3
9
0
```

the program displays the number 25.

Count On Me

"Write *I will not talk in class* on the blackboard 100 times."

What your teacher really meant was this:

```
Set the count to 0.
As long as the count is less than 100,
    Write 'I will not talk in class' on the blackboard,
    Add 1 to the count.
```

Fortunately, you didn't know about loops and counters at the time. If you pointed out all this stuff to your teacher, you'd have gotten into a lot more trouble than you were already in.

One way or another, life is filled with examples of counting loops. And computer programming mirrors life — or is it the other way around? When you tell a computer what to do, you're often telling the computer to print three lines, process ten accounts, dial a million phone numbers, or whatever. Because counting loops is so common in programming, the people who create programming languages

have developed statements just for loops of this kind. In Java, the statement that repeats something a certain number of times is called a *for statement.* Listings 6-2 and 6-3 illustrate the use of the `for` statement. Listing 6-2 has a rock-bottom simple example and Listing 6-3 has a more exotic example. Take your pick.

LISTING 6-2: **The World's Most Boring for Loop**

```
void main() {
    for (int count = 1; count <= 10; count++) {
        print("The value of count is ");
        print(count);
        println(".");
    }
    println("Done!");
}
```

Figure 6-3 shows you what you get when you run the program from Listing 6-2. (You get exactly what you deserve.) The `for` statement in Listing 6-2 starts by setting the `count` variable to 1. Then the statement tests to make sure that `count` is less than or equal to 10 (which it certainly is). Then the `for` statement dives ahead and executes the printing statements between the curly braces. (At this early stage of the game, the computer prints `The value of count is 1.`) Finally, the `for` statement does that last thing inside its parentheses — it adds 1 to the value of `count`.

```
The value of count is 1.
The value of count is 2.
The value of count is 3.
The value of count is 4.
The value of count is 5.
The value of count is 6.
The value of count is 7.
The value of count is 8.
The value of count is 9.
The value of count is 10.
Done!
```

FIGURE 6-3: Counting to ten.

With `count` now equal to 2, the `for` statement checks again to make sure that `count` is less than or equal to 10. (Yes, 2 is smaller than 10.) Because the test turns out okay, the `for` statement marches back into the curly-braced statements and prints `The value of count is 2` on the screen. Finally, the `for` statement does that last thing inside its parentheses — it adds 1 to the value of `count`, increasing the value of `count` to 3.

And so on. This whole business repeats until, after ten iterations, the value of `count` finally reaches 11. When this happens, the check for `count` being less than or equal to 10 fails and the loop's execution ends. The computer jumps to whatever statement comes immediately after the `for` statement. In Listing 6-2, the computer prints `Done!` as its output. Figure 6-4 illustrates the whole process.

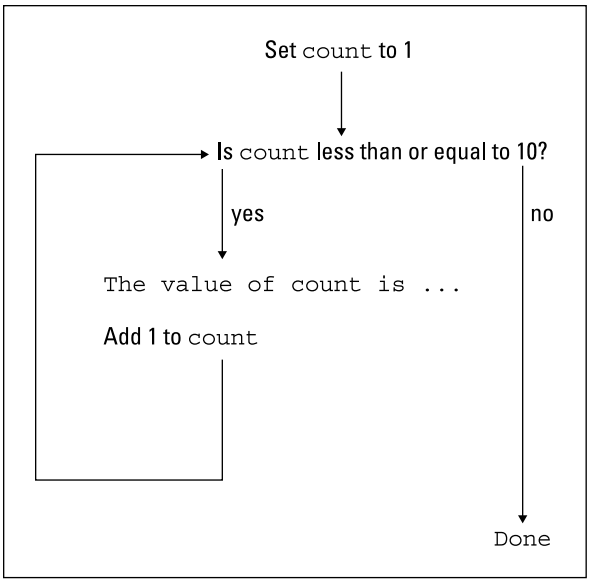


FIGURE 6-4: The action of the `for` loop in Listing 6-2.

The anatomy of a for statement

After the word *for*, you always put three things in parentheses. The first of these three is called an *initialization*, the second is an *expression*, and the third is an *update:*

```
for ( initialization ; expression ; update )
```

Each of the three items in parentheses plays its own, distinct role:

- The **initialization** is executed once, when the run of your program first reaches the `for` statement.
- The **expression** is evaluated several times (before each iteration).
- The **update** is also evaluated several times (at the end of each iteration).

If it helps, think of the loop as though its text is shifted all around:

```
int count = 1
for count <= 10 {
    print("The value of count is ");
    print(count);
    println(".");
    count++;
}
```

You can't write a real `for` statement this way. Even so, this is the order in which the parts of the statement are executed.

WARNING

If you declare a variable in the initialization of a `for` loop, you can't use that variable outside the loop. For instance, in Listing 6-2, you see an error message if you try putting `println(count)` after the end of the loop.

TIP

Anything that can be done with a `for` loop can also be done with a `while` loop. Choosing to use a `for` loop is a matter of style and convenience, not necessity.

The world premiere of "Al's All Wet"

Listing 6-2 is very nice, but the program in that listing does nothing interesting. For a more eye-catching example, see Listing 6-3. In Listing 6-3, I make good on a promise I make in Chapter 5: The program in Listing 6-3 prints all the lyrics of the hit single "Al's All Wet." (You can find the lyrics in Chapter 5.)

LISTING 6-3: **The Unabridged "Al's All Wet" Song**

```
void main() {
    String intro = """
            Al's all wet. Oh, why is Al all wet? Oh,
            Al's all wet 'cause he's standing in the rain.""";

    for (int verse = 1; verse <= 3; verse++) {
        println(intro);
        println("Why is Al out in the rain?");

        println(switch (verse) {
            case 1 -> "That's because he has no brain.";
            case 2 -> "That's because he is a pain.";
            case 3 -> "'Cause this is the last refrain.";
            default -> "";
        });
```

```
        switch (verse) {
            case 3:
                println("Last refrain, last refrain,");
            case 2:
                println("He's a pain, he's a pain,");
            case 1:
                println("Has no brain, has no brain,");
        }

        println("""
                In the rain, in the rain.
                Oh, oh, oh, oh""");
        println("");
    }

    println(intro);
}
```

Listing 6-3 is nice because it combines many of the ideas from Chapters 5 and 6. Listing 6-3 has two switches nested inside a `for` loop. One of them is a `switch` expression; the other is a `switch` statement with fall-through. As the value of the `for` loop's counter variable (`verse`) goes from `1` to `2` and then to `3`, Java executes all the parts in both of these switches. When the program is near the end of its run and execution has dropped out of the `for` loop, the program's last statement prints the song's final verse.

TECHNICAL STUFF

When I boldly declare that a `for` statement is for counting, I'm stretching the truth just a bit. Java's `for` statement is *versatile:* You can use a `for` statement in situations that have nothing to do with counting. For instance, a statement with no update part, such as `for (i = 0; i < 10; )`, just keeps going. The looping ends when some action inside the loop assigns a big number to the variable `i`. You can even create a `for` statement with nothing inside the parentheses. The loop `for ( ; ; )` runs forever, which is good if the loop controls a serious piece of machinery. Usually, when you write a `for` statement, you're counting how many times to repeat something. But, in truth, you can do just about any kind of repetition with a `for` statement.

TRY IT OUT

Would you like some practice? Try these experiments and challenges:

FOR AMOUR

A `for` statement's initialization may have several parts. A `for` statement's update may also have several parts. To find out how, enter the following lines in Java's JShell, or add the lines to a small Java program:

```
for (int i = 0, j = 10; i < j; i++, j--) {System.out.println(i + " " + j);}
```

In this code, I recommend calling `System.out.println` instead of plain old `println`. Where I sit in the year 2024, JShell still doesn't accept the shorter `println` call. One way or another, `System.out.println` always works.

COLLECTING VALUES

What's the output of the following code?

```
int total = 0;
for (int i = 0; i < 10; i++) {
    total += i;
}
System.out.println(total);
```

In this code, the variable `total` is called an *accumulator* because it accumulates (adds up) a bunch of values inside the loop.

FACTORIAL

In mathematics, the exclamation point (!) means *factorial* — the number you get when you multiply all the positive `int` values up to and including a certain number. For example, 3! is $1 \times 2 \times 3$, which is 6. And 5! is $1 \times 2 \times 3 \times 4 \times 5$, which is 120.

Write a program in which the user types a positive `int` value (call it *n*), and Java displays the value of *n*! as its output.

SEEING STARS

Without running the following code, try to predict the code's output:

```
for (int row = 0; row < 5; row++) {
    for (int column = 0; column < 5; column++) {
        print("*");
    }
    println();
}
```

After making your prediction, run the code to find out whether your prediction is correct.

SEEING MORE AND MORE STARS

The code in this experiment is a slight variation on the code in the previous experiment. First, try to predict what the code will output. Then run the code to find out whether your prediction is correct:

```
for (int row = 0; row < 5; row++) {
    for (int column = 0; column <= row; column++) {
        print("*");
    }
    println();
}
```

THREE TRIANGLES

Write a program that uses loops to display three copies of the following pattern, one after another:

```
*****
****
***
**
*
```

You Can Always Get What You Want

Fools rush in where angels fear to tread.

—Alexander Pope

Today, I want to be young and foolish (or, at the very least, foolish). Look back at Figure 6-2 and notice how Java's `while` loop works. When execution enters a `while` loop, the computer checks to make sure that the loop's condition is true. If the condition isn't true, the statements inside the loop are never executed — not even once. In fact, you can easily cook up a `while` loop whose statements are never executed (although I can't think of a reason you would ever want to do it):

```
int twoPlusTwo = 2 + 2;
while (twoPlusTwo == 5) {
    println("""
        Are you kidding?
        2 + 2 doesn't equal 5.
        Everyone knows that 2 + 2 equals 3.""");
}
```

In spite of this silly `twoPlusTwo` example, the `while` statement turns out to be the most versatile of Java's looping constructs. In particular, the `while` loop is good for situations in which you must look before you leap. For example, "While money

is in my account, write a mortgage check every month." When you first encounter this statement, if your account has a zero balance, you don't want to write a mortgage check — not even a single check.

But at times (not many), you want to leap before you look. Take, for instance, the situation in which you're asking the user for a response. Maybe the user's response makes sense, but maybe it doesn't. If it doesn't, you want to ask again. Maybe the user's finger slipped, or perhaps the user didn't understand the question.

Figure 6-5 shows some runs of a program to delete a file. Before deleting the file, the program asks the user whether making the deletion is okay. If the user answers *y* or *n,* the program proceeds according to the user's wishes. But if the user enters any other character (any digit, uppercase letter, punctuation symbol, or whatever), the program asks the user for another response.

```
Delete evidence? (y/n) n
Sorry, buddy. Just asking.


Delete evidence? (y/n) u
Delete evidence? (y/n) Y
Delete evidence? (y/n) L
Delete evidence? (y/n) 8
Delete evidence? (y/n) .
Delete evidence? (y/n) y
Okay, here goes...
The evidence has been deleted.
```

FIGURE 6-5: Two runs of the code in Listing 6-4.

To write this program, you need a loop — a loop that repeatedly asks the user whether the file should be deleted. The loop keeps asking until the user gives a meaningful response. Now, the thing to notice is that the loop doesn't need to check anything before asking the user the first time. Indeed, before the user gives the first response, the loop has nothing to check. The loop doesn't start with "As long as such-and-such is true, then get a response from the user." Instead, the loop just leaps ahead, gets a response from the user, and then checks the response to see whether it makes sense.

That's why the program in Listing 6-4 has a *do* loop (also known as a *do . . . while* loop). With a `do` loop, the program jumps right in, takes action, and then checks a condition to see whether the result of the action makes sense. If the result makes sense, execution of the loop is done. If not, the program goes back to the top of the loop for another go-round.

LISTING 6-4: **To Delete or Not to Delete**

```
import java.io.File;

void main() {
    var evidence = new File("cookedBooks.txt");
    char reply;

    do {
        reply = readln("Delete evidence? (y/n) ").charAt(0);
    } while (reply != 'y' && reply != 'n');

    if (reply == 'y') {
        println("Okay, here goes...");
        evidence.delete();
        println("The evidence has been deleted.");
    } else {
        println("Sorry, buddy. Just asking.");
    }
}
```

Figure 6-5 shows two runs of the code in Listing 6-4. The program accepts lowercase letters *y* and *n*, but not the uppercase letters *Y* and *N*. To make the program accept uppercase letters, change the conditions in the code as follows:

```
} while (reply != 'y' && reply != 'Y' && reply != 'n' && reply!='N');
```

Figure 6-6 shows the flow of control in the loop of Listing 6-4. With a `do` loop, the situation in the `twoPlusTwo` program (shown at the beginning of this section) can never happen. Because the `do` loop carries out its first action without testing a condition, every `do` loop is guaranteed to perform at least one iteration.

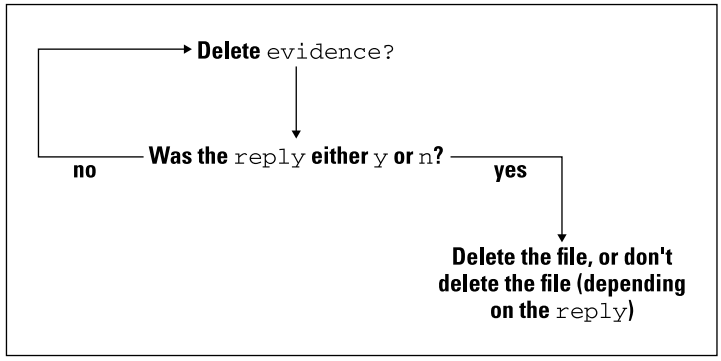


FIGURE 6-6: Here we go loop, do loop.

Who moved my file?

The location of Listing 6-4's `cookedBooks.txt` file on your computer's hard drive depends on several factors. If you create a `cookedBooks.txt` file in the wrong directory, the code in Listing 6-4 cannot delete your file. (More precisely, if `cookedBooks.txt` is in the wrong directory on your hard drive, the code in Listing 6-4 can't find the `cookedBooks.txt` file in preparation for deleting the file.)

In most settings, you start testing Listing 6-4 by creating a project within your IDE. The new project lives in a folder on your hard drive. That folder is called the project's *root folder*. With the code in Listing 6-4, the `cookedBooks.txt` file probably belongs directly inside your project's root folder.

For example, I have a project named `06-04`. That project lives on my hard drive in a folder named `06-04` — the project's root folder. Directly inside that `06-04` root folder, I have a file named `cookedBooks.txt`. Also, inside the `06-04` folder, I have a `src` subfolder. The `src` subfolder contains my `DeleteEvidence.java` file.

TIP

If putting `cookedBooks.txt` in the project's root folder doesn't work, try putting it in the folder that contains the code in Listing 6-4. And, if all else fails, add the following code to Listing 6-4 immediately after the `new File` statement:

```
try {
    println("Looking for " + evidence.getCanonicalPath());
} catch (java.io.IOException e) {
    e.printStackTrace();
}
```

When you run the code, Java tells you where, on your hard drive, the `cookedBooks.txt` file should be.

CROSS REFERENCE

For more information about files and their folders, see Chapter 9.

Reading a single character

The `readln` method grabs a whole line of keyboard input and delivers it to your Java program. As far as Java is concerned, this input is a `String` instance. That's okay for some programs, but in Listing 6-4, you're not interested in reading a whole line of text. You expect the user to press one letter — either `y` or `n`. So, in Listing 6-4, I don't create a `String` variable to store the user's response. Instead, I create a `char` variable — a variable that stores just one symbol at a time.

The `reply` variable in Listing 6-4 has type `char`. So I have to find a way to stuff the big `readln` string into the `reply` variable's tiny `char`. I do this by calling the `String` class's `charAt` method. When you apply `charAt` to a string, you get a single `char` value. See Table 6-1.

TABLE 6-1 The String class's charAt method

The value of `myString`	Expression containing `charAt`	The Expression's Value
`"yes"`	`myString.charAt(0)`	`'y'`
`"no"`	`myString.charAt(0)`	`'n'`
`"Yes"`	`myString.charAt(0)`	`'Y'` (which is not equal to `'y'`)
`"yes"`	`myString.charAt(1)`	`'e'`
`"yes"`	`myString.charAt(2)`	`'s'`
`"yes"`	`myString.charAt(3)`	(Bad method call)

REMEMBER

A `String` variable can contain many characters or just one. But a `String` variable that contains only one character isn't the same as a `char` variable. No matter what you put in a `String` variable, `String` variables and `char` variables have to be treated differently.

On the var side

The import declaration at the top of Listing 6-4 lets you abbreviate `java.io.File` in the rest of the code with the simple name `File`. Given that import declaration, here's one way to associate the made-up name `evidence` with an instance of Java's `File` class:

```
File evidence = new File("cookedBooks.txt");
```

In this statement, the first two words, `File evidence`, reserve the variable name `evidence` to refer to an instance of Java's `File` class, and the expression `new File("cookedBooks.txt")` stands for a brand-new instance of the `File` class.

Around 2018, the stewards of Java decided that statements of this kind are too wordy. Repeating the word `File` is redundant and unnecessary. So, with the release of Java 10, they created a new word: `var`. They said that, in certain situations, `var` can replace the class name's first occurrence:

```
var evidence = new File("cookedBooks.txt");
```

In this statement, the word `var` tells Java to figure out on its own that `evidence` refers to a `File` instance. Believe it or not, older versions of Java couldn't jump to that conclusion. Before Java 10 came along, repeating a word like `File` was the only option.

This newfangled `var` word is handy. But remember, you can't always replace `File evidence` with `var evidence`. For example, you can write this:

```
File evidence;                              // This is okay.
evidence = new File("cookedBooks.txt");
```

But you can't write this:

```
var evidence;                               // Don't do this.
evidence = new File("cookedBooks.txt");
```

Every `var` declaration must include an initialization. When Java sees the line

```
var evidence;                               // Don't do this.
```

with no initialization, Java says, "Cannot infer type: 'var' on variable without initializer." That means Java isn't willing to look at other statements to find out whether `var evidence` should stand for `File evidence`, `String evidence`, `Random evidence`, or whatever.

For more talk about when you can and cannot use `var`, see Chapter 7.

CROSS REFERENCE

The expression `new File("cookedBooks.txt")` is an example of a constructor call. For the lowdown on constructor calls, see Chapter 8.

File handling in Java

In Listing 6-4, the actual file-handling statements deserve some attention. These statements involve the use of classes, objects, and methods. Many of the meaty details about these things are in other chapters, like Chapter 9. Even so, I can't do any harm by touching on some highlights right here.

In Listing 6-4, the `main` method starts with the statement

```
var evidence = new File("cookedBooks.txt");
```

Oddly enough, this statement doesn't create a new file on your computer's hard drive. Instead, this statement creates a new object in your Java program. This object, formed from the `File` class, describes everything the program needs to

know about a file that's supposedly on your hard drive. If you really do have a file named `cookedBooks.txt`, and if that file lives in the folder where Java expects to find it, then the variable `evidence` refers to information about that disk file.

The `evidence` object, as an instance of the `File` class, has a `delete` method. (What can I say? It's in the API documentation.) When you call `evidence.delete`, the Java program tries to delete the file from your hard drive.

Of course, you can't delete something that doesn't already exist. Even Al Capone wouldn't bother to delete evidence that doesn't exist. To avoid some needless method calling, you have a few options. The simplest is to call the `File` class's `exists` method.

```
import java.io.File;

void main() {
    var evidence = new File("cookedBooks.txt");
    if (evidence.exists()) {
        // The rest of the code goes here.
    }
}
```

When you call `evidence.exists()`, the method looks in the folder where Java expects to find `cookedBooks.txt`. The value of `evidence.exists()` is `true` if Java finds `cookedBooks.txt` inside that folder. Otherwise, `evidence.exists()` is `false`.

Block on the while side

A bunch of statements surrounded by curly braces forms a block. (See Chapter 5.) If you declare a variable inside a block, you generally can't use that variable outside the block. For instance, in Listing 6-4, you see some error messages if you make the following change:

```
do {
    char reply = readln("Delete evidence? (y/n) ").charAt(0);
} while (reply != 'y' && reply != 'n');

if (reply == 'y')
    // ... Etc.
```

With the declaration `char reply` inside the loop's curly braces, no use of the name `reply` makes sense anywhere outside the braces. When you try to compile this code, you get three error messages — two for the `reply` words in `while (reply != "y" && reply != "n")` and a third for the `if` statement's `reply`.

So, in Listing 6-4, your hands are tied. The program's first real use of the `reply` variable is inside the block's curly braces. But to make that variable available outside the braces, you have to declare `reply` before the start of the block. In this situation, you're best off declaring the `reply` variable without initializing the variable. Very interesting!

CROSS REFERENCE

To read more about variable initializations, see Chapter 4.

All versions of Java have the three kinds of loops described in this chapter (`while` loops, `for` loops, and `do ... while` loops). But newer Java versions (namely, Java 5 and beyond) have yet another kind of loop, called an *enhanced for loop.* For a look at Java's enhanced `for` loop, see Chapters 12 and 13.

TRY IT OUT

MISSED OPPORTUNITY

Copy the code from Listing 6-1, but with the following change:

```
String input = readln("Pick a number from 1 to 10: ");
int inputNumber = Integer.parseInt(input);
numGuesses++;

do {
    println("");
    println("Try again...");
    input = readln("Pick a number from 1 to 10: ");
    inputNumber = Integer.parseInt(input);
    numGuesses++;
} while (inputNumber != randomNumber);

print("You win after ");
println(numGuesses + " guesses.");
```

The code in Listing 6-1 has a `while` loop, but this modified code has a `do` loop. Does this modified code work correctly? Why or why not?

LET'S BUST OUT OF HERE

In Chapter 5, you use `break` statements to jump out of a `switch`. But a `break` statement can also play a role inside a loop. To find out how it works, run a program containing the following code:

```
while (true) {
    int i = Integer.parseInt(readln("Enter an int value: "));
    if (i == 0) {
```

```
        break;
    }
    println(i);
}
println("Done!");
```

The loop's condition is always `true`. It's like starting a loop with the line

```
while (1 + 1 == 2)
```

If it weren't for the `break` statement, the loop would run forever. Fortunately, when you execute the `break` statement, Java jumps to the code immediately after the loop.

CARRY ON AND KEEP CODING

In addition to its `break` statement, Java has a `continue` statement. When you execute a `continue` statement, Java skips to the end of its loop and begins the next iteration of that loop. To see it in action, run a program containing the following code:

```
while (true) {
    int i = Integer.parseInt(readln("Enter an int value: "));
    if (i > 10) {
        continue;
    }
    if (i == 0) {
        break;
    }
    println(i);
}
println("Done!");
```

3 Working with the Big Picture: Object-Oriented Programming

IN THIS PART . . .

Find out what classes and objects are (without bending your brain out of shape).

Be the architect of your virtual world by constructing brand-new objects.

Deal with data that's stored on your disk.

Learn how to reuse existing code (saving time and money).

IN THIS CHAPTER

- Thinking like a real-life object-oriented programmer
- Passing values to and from methods
- Hiding details in your object-oriented code

Chapter 7
The Inside scOOP

As a computer book author, I've been told, over and over again: Don't expect people to read sections and chapters in their logical order. People jump around, picking what they need and skipping what they don't feel like reading. With that in mind, I realize that you may have skipped Chapter 1. If that's the case, please don't feel guilty. You can compensate in just 60 seconds by reading the following information, culled from Chapter 1:

> *Because Java is an object-oriented programming language, your primary goal is to describe classes and objects. A class is the idea behind a certain kind of thing. An object is a concrete instance of a class. First, you define a class. Then, from the class definition, you construct individual objects.*

Of course, you can certainly choose to skip over the 60-second summary paragraph. If that's the case, you may want to recoup some of your losses. You can do that by reading the following two-word summary of Chapter 1:

> *Classes; objects.*

Defining a Class (What It Means to Be an Account)

What distinguishes one bank account from another? If you ask a banker this question, you hear a long sales pitch. The banker describes interest rates, fees, penalties — the whole routine. Fortunately for you, I'm not interested in all that. Instead, I want to know how my account is different from your account. After all, my account is named *Barry Burd, trading as Burd Brain Consulting,* and your account is named *Jane Q. Reader, trading as Budding Java Expert.* My account has $24.02 in it. How about yours?

When you come right down to it, the differences between one account and another can be summarized as values of variables. Maybe there's a variable named `balance`. For me, the value of `balance` is `24.02`. For you, the value of `balance` is `55.63`. The question is, when writing a computer program to deal with accounts, how do I separate my `balance` variable from your `balance` variable?

The answer is to create two separate objects. Let one `balance` variable live inside one object and let the other `balance` variable live inside the other object. While you're at it, put a `name` variable and an `address` variable in each of the objects. And there you have it: two objects, and each object represents an account. More precisely, each object is an instance of the `Account` class. (See Figure 7-1.)

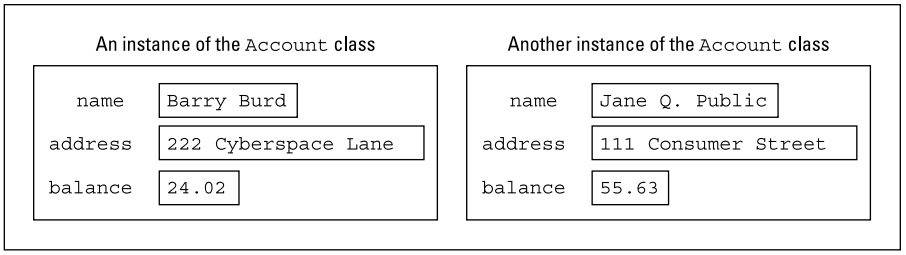


FIGURE 7-1: Two objects.

REMEMBER

The words *object* and *instance* have the same meaning. An `Account` class is like a blueprint from which you build objects. Each object that you build using the `Account` class is called an *instance* of the `Account` class.

Figure 7-1 has two names, two addresses, and two balances. So I ask, in a Java program, how do you refer to my balance as opposed to your balance? Well, you have two objects sitting around, so maybe you have variables to refer to these two objects. Create one variable named `myAccount` and another variable named

yourAccount. The myAccount variable refers to my object (my instance of the Account class) with all the stuff that's inside it. To refer to my balance, write

```
myAccount.balance
```

To refer to my name, write

```
myAccount.name
```

Then yourAccount.balance refers to the value in your object's balance variable, and yourAccount.name refers to the value of your object's name variable. To tell Java how much I have in my account, you can write

```
myAccount.balance = 24.02;
```

For want of any better terminology, this way of referring to an object's values is called *dot notation*. To display your name on the screen, you can write

```
println(yourAccount.name);
```

These ideas come together in Listings 7-1 and 7-2. Here's Listing 7-1:

LISTING 7-1: What It Means to Be an Account

```
class Account {
    String name;
    String address;
    double balance;
}
```

The Account class in Listing 7-1 defines what it means to be an Account. In particular, Listing 7-1 tells you that each of the Account class's instances has three variables: name, address, and balance. This is consistent with the information in Figure 7-1. Java programmers have a special name for variables of this kind (variables that belong to instances of classes). Each of these variables — name, address, and balance — is called a *field.*

REMEMBER

A variable declared inside a class but not inside any particular method is a *field.* In Listing 7-1, the variables name, address, and balance are fields.

If you've been grappling with the material in earlier chapters, the code for the class Account in Listing 7-1 may come as a surprise to you. Can you really define a complete Java class with only four lines of code (give or take a curly brace)? You

certainly can. A *class* is a grouping of existing things. In the `Account` class of Listing 7-1, those existing things are two `String` values and a `double` value.

WARNING

Your IDE won't offer to run the code in Listing 7-1. That's because Listing 7-1 has no `main` method. The run of every Java program must begin with the `main` method, so the code in Listing 7-1 has no starting point. To use the listing's `Account` class, your project must contain at least two Java files: one file with the code in Listing 7-1 and another file with a `main` method. In this chapter, Listing 7-2 has a handy-dandy `main` method, so the `Account` class in Listing 7-1 won't go to waste.

Declaring variables and creating objects

A young fellow approaches me while I'm walking down the street. He tells me to print the value of 2 + 2 so I print the number 4. If you must know, I print it with chalk on the sidewalk. But where I print the number 4 doesn't matter. What matters is that some guy issues instructions, and I follow the instructions.

Later that day, an elderly woman sits next to me on a park bench. She says, "An account has a name, an address, and a balance." And I say, "That's fine, but what do you want me to do about it?" In response, she just stares at me, so I don't do anything about her *account* pronouncement. I just sit there, she sits there, and we both do absolutely nothing.

Listing 7-1, shown earlier, is like the elderly woman. This listing defines what it means to be an `Account`, but the listing doesn't tell me to do anything with my account, or with anyone else's account. In order to do something, I need a second piece of code. I need a `main` method. Fortunately, while the woman and I sit quietly on the park bench, a young child comes by with Listing 7-2.

LISTING 7-2: Dealing with Account Objects

```
void main() {
    var myAccount = new Account();
    var yourAccount = new Account();

    myAccount.name = "Barry Burd";
    myAccount.address = "222 Cyberspace Lane";
    myAccount.balance = 24.02;

    yourAccount.name = "Jane Q. Public";
    yourAccount.address = "111 Consumer Street";
    yourAccount.balance = 55.63;
```

```
        println(myAccount.name + " (" + myAccount.address + ") " +
                "has $" + myAccount.balance);
        println(yourAccount.name + " (" + yourAccount.address + ") " +
                "has $" + yourAccount.balance);
}
```

Taken together, Listings 7-1 and 7-2 form one complete program. The code in Listing 7-2 defines a main method with variables of its own — `myAccount` and `yourAccount`.

When Java performs the initialization `myAccount = new Account()`, Java creates a new object (a new instance of the `Account` class) and makes the variable `myAccount` refer to that new object. Figure 7-2 illustrates the situation.

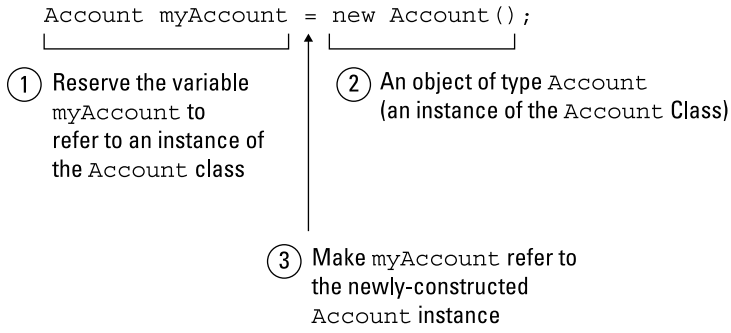

Account myAccount = new Account();
1 Reserve the variable myAccount to refer to an instance of the Account class
2 An object of type Account (an instance of the Account Class)
3 Make myAccount refer to the newly-constructed Account instance

FIGURE 7-2: The making of `myAccount`.

CROSS REFERENCE

Technically, when your program executes `new Account()`, Java creates an object by calling the `Account` class's *constructor*. When you see Java's `new` keyword, think "constructor call." I have a lot more to say about constructors and constructor calls in Chapter 8.

Using an object's fields

After you've bitten off and chewed the `main` method's first two lines, the rest of the code in Listing 7-2 is sensible and straightforward. You have three statements that put values in the `myAccount` object's fields, three statements that put values in the `yourAccount` object's fields, and two statements that do some printing. Figure 7-3 shows the program's output.

FIGURE 7-3: Running the code in Listings 7-1 and 7-2.

```
Barry Burd (222 Cyberspace Lane) has $24.02
Jane Q. Public (111 Consumer Street) has $55.63
```

At this point, I have a confession to make. When I was first learning Java, I sometimes wrote code like this:

```
// Bad code!
Account myAccount;
myAccount.name = "Barry";
myAccount = new Account();
```

Separating `Account myAccount` from `new Account()` was okay. But referring to `myAccount.name` before doing `new Account()` was not okay. When I tried to compile this code, I got the message: `variable myAccount might not have been initialized`. Before you do `new Account()`, you can't assign a value to an object's `name` field. You can't do it because the object doesn't exist yet.

Some people read `Account myAccount` as if it's supposed to mean "`myAccount` is an `Account`" or "the variable `myAccount` refers to an instance of the `Account` class." That's not really what `Account myAccount` means. Instead, `Account myAccount` means, "If and when I make the variable `myAccount` refer to something, that something will be an instance of the `Account` class." So, what's the difference?

The difference is that simply declaring `Account myAccount` doesn't make the `myAccount` variable refer to an object. All it does is reserve the variable name `myAccount` so that the name can eventually refer to an instance of the `Account` class. The creation of an actual object doesn't come until Java executes `new Account()`.

One program; several classes

Each program in Chapters 3–6 consists of a single class. That's great for a book's introductory chapters. But in real life, a typical program consists of hundreds or even thousands of classes. The program that spans Listings 7-1 and 7-2 consists of two classes. Sure, having two classes isn't like having thousands of classes, but it's a step in that direction.

In practice, most programmers put each class in a file of its own. When you create a program, such as the one in Listings 7-1 and 7-2, you create two files on your computer's hard drive. Therefore, when you download this section's example from the web, you get two separate files — `Account.java` and `UseAccount.java`.

For information about running a program consisting of more than one `.java` file in Eclipse, NetBeans, IntelliJ IDEA, and VS Code, visit this book's website (`http://javafordummies.allmycode.com`).

The road not taken

If you want, you can initialize the fields in a class declaration. For example, you can write

```
class Account {
    String name = "";
    String address = "(Unknown)";
    double balance = 0.00;
}
```

Then, after you've created an instance of the `Account` class, you can override some or all of the initial values:

```
    yourAccount.name = "Jane Q. Public";
    yourAccount.balance = 55.63;
```

But there's something you definitely can't do. Chapter 6 tells you that when you initialize a variable, you can replace the variable's type name with the word `var`. In other words, you can write

```
void main() {
    var evidence = new File("cookedBooks.txt");   //Okay!
    var reply = 'n';                              //Okay!
    //... and so on.
```

Yes, this rule about `var` applies to the variables that you declare inside your method bodies. But it *doesn't* apply to a class's fields.

I illustrate this concept with a few lines of code:

```
class Account {
    String name;                    // This is OK.
    String address = "(Unknown)";   // This IS OK.
    var balance = 0.00;             // Not OK. It should be 'double'.
                                    // Forgive me!
}
```

Honestly, it makes me nervous to type that `var balance` line, even while I warn you that the line is incorrect. When you create a class's field, you can initialize the field or not. That's up to you. But you can't use `var` in the field's declaration.

TRY IT OUT

It's your turn to create some Java classes.

GETTING ORGANIZED

An `Organization` has a name (such as *XYZ Company*), an annual revenue (such as $100,000.00), and a `boolean` value indicating whether the organization is or is not a profit-making organization. Companies that manufacture and sell products are generally profit-making organizations; groups that provide aid to victims of natural disasters are generally not profit-making organizations.

Declare your own `Organization` class. Declare a main method that creates organizations and displays information about those organizations.

YUMMY FOODS

A product for sale in a food store has several characteristics: a type of food (peach slices), a weight (500 grams), a cost ($1.83), a number of servings (4), and a number of calories per serving (70).

Declare a `FoodProduct` class. Declare a `main` method that creates `FoodProduct` instances and displays information about those instances.

IMPULSE BUYING

Try to run the following code (and make sure you understand why it doesn't run).

```
class Purchase {
    String itemName;
    int quantity;
}

void main() {
    Purchase purchase;
    purchase.itemName = "book";
    purchase = new Purchase();
    println(purchase.itemName);
}
```

Defining a Method within a Class (Displaying an Account)

Imagine a table containing the information about two accounts. (If you have trouble imagining such a thing, just look at Table 7-1.)

TABLE 7-1 A Table of Accounts

Name	Address	Balance
Barry Burd	222 Cyberspace Lane	24.02
Jane Q. Public	111 Consumer Street	55.63

You can think of Listings 7-1 and 7-2 this way:

- Listing 7-1 creates the table's header row.
- Most of Listing 7-2 creates the table's two body rows.
- In Listing 7-2, the two `println` calls make use of the information in the table.

The table's header represents the `Account` class, and the table's rows represent objects — instances of the `Account` class.

Chapter 1 describes a class that keeps track of employees in a small company. Here's a quotation from that chapter:

> *An `Employee` has data (the employee's name, hours per week, and pay), but an `Employee` also has code for working with the data (`add1Hour` and `add50Dollars`). In object-oriented programming, each class owns code for working with that class.*

In Table 7-1, each account has three things — a name, an address, and a balance. But, with object-oriented programming, each account can have a name, an address, a balance, and a way of being displayed.

In object-oriented programming, each object has its own built-in functionality. An account knows how to display itself. A string can tell you whether it has the same characters inside it as another string. An instance of `Random` can generate an `int` value for you. In object-oriented programming, each object has its own methods. These methods are little subprograms you can call to have an object do things to (or for) itself.

And why is this a good idea? It's good because you're making pieces of data take responsibility for themselves. With object-oriented programming, all the functionality that's associated with an account is collected inside the code for the `Account` class. Everything you have to know about a string is located in the file `String.java`. Anything having to do with year numbers (whether they have two or four digits, for instance) is handled right inside the `Year` class. Therefore, if anybody has problems with your `Account` class or your `Year` class, they know just where to look for all the code. That's great!

Imagine an enhanced Account table. In this new table, each object has built-in functionality. Each account knows how to display itself on the screen. Each row of the table has its own copy of a `display` method. Of course, you don't need much imagination to picture this table. I just happen to have a table you can look at. It's Table 7-2.

TABLE 7-2 A Better Account Class

Name	Address	Balance	Display
Barry Burd	222 Cyberspace Lane	24.02	`println ...`
Jane Q. Public	111 Consumer Street	55.63	`println ...`

An account that displays itself

In Table 7-2, each account object has four things — a name, an address, a balance, and a way of displaying itself on the screen. Listings 7-3 and 7-4 show programs that implement the ideas in Table 7-2.

LISTING 7-3: An Account Displays Itself

```
import java.io.IO;

class Account {
    String name;
    String address;
    double balance;

    void display() {
        IO.print(name + " (" + address + ") " + "has $" + balance);
    }
}
```

LISTING 7-4: Using the Improved Account Class

```
void main() {
    var myAccount = new Account();
    var yourAccount = new Account();

    myAccount.name = "Barry Burd";
    myAccount.address = "222 Cyberspace Lane";
    myAccount.balance = 24.02;
```

```
    yourAccount.name = "Jane Q. Public";
    yourAccount.address = "111 Consumer Street";
    yourAccount.balance = 55.63;

    myAccount.display();
    println("");
    yourAccount.display();
    println("");
}
```

A run of the code in Listings 7-3 and 7-4 looks just like a run of Listings 7-1 and 7-2. (Refer to Figure 7-3.)

In Listing 7-3, the `Account` class has four things in it: three fields (`name`, `address`, and `balance`) and a `display` method. Taken together, a class's fields and methods are called *members* of the class. The `Account` class's members match up with the four columns in Table 7-2.

Notice how you refer to members of the `Account` class:

- To refer to the `myAccount` object's `name`, you write

  ```
  myAccount.name
  ```

- To make the `myAccount` object display itself on the screen, you write

  ```
  myAccount.display()
  ```

  Like the `name` reference, this `display` reference uses dot notation. The only difference is that calling a method uses parentheses, and referring to a field doesn't use parentheses.

- Inside the `Account` class declaration, you refer to members without using dot notation.

  For example, in Listing 7-3, the members `name`, `address`, and `balance` come without dots in the statement `IO.print(name + " (" + address + ") " + "has $" + balance)`.

Inside a class declaration, you have the option of referring to members using dot notation and Java's `this` keyword:

```
IO.print(this.name + " (" + this.address + ") " + "has $" + this.balance);
```

For more information about Java's `this` keyword, see Chapter 10.

The display method's header

In Listing 7-3, the `display` method's declaration has a header and a body. (See Chapter 3.) The header consists of two words and some parentheses, like so:

```
void display()
```

Let's dissect this method's header:

- **The word *void* tells Java that when the `display` method is called, the `display` method doesn't return anything to the place that called it.**

  Java's `main` method doesn't return anything to whatever environment called it. That's why the `main` method's header always starts with the word `void`.

  In case you're wondering, the next section has a method that returns something to the place that called it.

- **The word `display` is the method's name.**

  If the method didn't have a name, you'd have no way of calling the method in Listing 7-4.

- **The parentheses contain all the things you're going to pass to the method when you call it.**

  When you call a method, you can pass information to that method on the fly. The `display` method in Listing 7-3 looks strange because the parentheses in the method's header have nothing inside them. This nothingness indicates that no information is passed to the `display` method when you call it. For a meatier example, see the next section.

The next section's example makes the `Account` class even more able to take care of itself.

AN IMPORTANT IMPORT

Feast your eyes on the import declaration at the top of Listing 7-3. Why do I include this import declaration? I call `print` without an import declaration in previous chapters' examples. Why do I bother importing `print` in Listing 7-3? The following quotation from Chapter 5 helps to unravel this riddle:

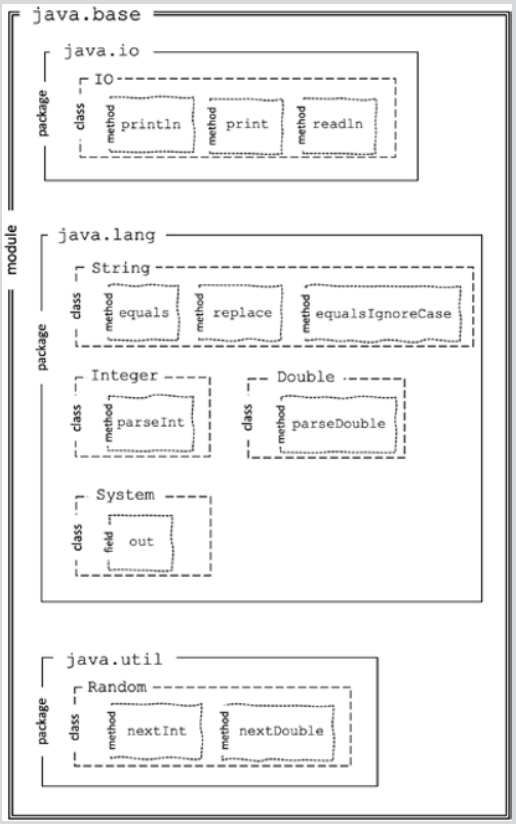
> *With each example in Chapters 1–6 of this book, you don't need an import declaration for any class inside* `java.base`. *Java automatically imports all these classes for you.*

Chapter 5 calls this an *implicit import declaration*. Java behaves as if your file starts with

```
import module java.base;
```

even though that declaration doesn't appear explicitly in your program's text.

What Chapter 5 says about implicit imports is perfectly true, but the chapter glosses over many details. This sidebar fills in some details. The details can be confusing, so feel free to skip over them and come back to this sidebar later. Anyway, as you read the following bullets, refer frequently to this sidebar's figure. Here we go:



(continued)

(continued)

- **One way to categorize Java classes is to divide them into two kinds: *explicit* and *implicit*.**

  Listings 7-1 and 7-3 contain *explicit* `Account` classes. But Listings 7-2 and 7-4 have `main` methods that don't appear to be inside any class's code. These `main` methods live inside *implicit* classes. A file that contains one of these implicit classes is called a *simple source file*.

- **The automatic importing for `java.base` applies only to simple source files.**

  Listing 7-3 has an explicit `Account` class, so Listing 7-3 doesn't implicitly import the classes in `java.base`. In particular, Listing 7-3 doesn't implicitly import the classes in `java.io.IO`. In Listing 7-3, you can call `IO.print`, but only if you explicitly import `java.io.IO` at the top of the file.

  In summary, a simple source file has an implicit class and the file implicitly imports `java.base`. A file that's not a simple source file has an explicit class and doesn't implicitly import `java.base`.

Here are a couple more (slightly esoteric) facts about implicit imports:

- **For the most part, the implicit importing for `java.base` applies only to classes, not to methods or fields.**

  In Chapters 5 and 6, you abbreviate `java.lang.Integer` with the short name `Integer`. That's because `java.lang.Integer` is the name of a class. (See this sidebar's figure.) The same reasoning applies when you abbreviate the class name `java.util.Random` with the short name `Random`.

  But `parseInt` and `nextInt` are methods, not classes. (See this sidebar's figure.) So, you can't write `parseInt` or `nextInt` alone in any of your code. Instead, you have to write `Integer.parseInt` and `Random.nextInt`.

- **In a simple source file, the implicit importing extends to methods inside the `java.io` package.**

  In Chapters 5 and 6, and in this chapter's Listing 7-4, I call `println("")` instead of `IO.println` or `java.io.IO.println("")`. I can't get away with that in Listing 7-3 because Listing 7-3 isn't a simple source file.

If you find this overwhelming, don't worry. This stuff takes time to learn. And remember, you can often get help from your IDE. In your IDE of choice, look for menus such as Code ➪ Optimize Imports or Source ➪ Organize Imports.

SHOW OFF

In the previous section, you create `Organization` and `FoodProduct` classes. Add `display` methods to both of these classes and create `main` methods that make use of these `display` methods.

Sending Values to and from Methods (Calculating Interest)

Think about sending someone to the supermarket to buy bread. When you do this, you say, "Go to the supermarket and buy some bread." (Try it at home. You'll have a fresh loaf of bread in no time at all!) Of course, at some other time, you send that same person to the supermarket to buy bananas. You say, "Go to the supermarket and buy some bananas." And what's the point of all this? Well, you have a method, and you have some on-the-fly information that you pass to the method when you call it. The method is named *goToTheSupermarketAndBuySome.* The on-the-fly information is either *bread* or *bananas,* depending on your culinary needs. In Java, the method calls would look like this:

```
goToTheSupermarketAndBuySome(bread);
goToTheSupermarketAndBuySome(bananas);
```

The things in parentheses are called *parameters,* or *parameter lists.* With parameters, your methods become much more versatile. Rather than buy the same item each time, you can send somebody to the supermarket to buy bread one time, bananas another time, and birdseed the third time. When you call your `goToTheSupermarketAndBuySome` method, you decide right then and there what you'll ask your pal to buy.

And what happens when your friend returns from the supermarket? "Here's the bread you asked me to buy," says your friend. By carrying out your wishes, your friend returns something to you. You make a method call, and the method returns information (or a loaf of bread).

The thing returned to you is called the method's *return value.* The general type of thing that's returned to you is called the method's *return type.* These concepts are made concrete in Listings 7-5 and 7-6.

LISTING 7-5: An Account That Calculates Its Own Interest

```
import java.io.IO;

class Account {
    String name;
    String address;
    double balance;

    void display() {
        IO.print(name + " (" + address + ") " + "has $" + balance);
    }

    double getInterest(double percentageRate) {
        return balance * percentageRate / 100.00;
    }
}
```

LISTING 7-6: Calculating Interest

```
void main() {
    var myAccount = new Account();
    var yourAccount = new Account();

    myAccount.name = "Barry Burd";
    myAccount.address = "222 Cyberspace Lane";
    myAccount.balance = 24.02;

    yourAccount.name = "Jane Q. Public";
    yourAccount.address = "111 Consumer Street";
    yourAccount.balance = 55.63;

    myAccount.display();
    println(" plus $" + myAccount.getInterest(5.00) + " interest");

    yourAccount.display();
    double yourInterestRate = 7.00;
    double yourInterestAmount = yourAccount.getInterest(yourInterestRate);
    println(" plus $" + yourInterestAmount + " interest");
}
```

Figure 7-4 shows the output of the code in Listings 7-5 and 7-6. In Listing 7-5, the `Account` class has a `getInterest` method. This `getInterest` method is called twice from the `main` method in Listing 7-6; the actual account balances and interest rates are different each time:

- **In the first call, the balance is 24.02 and the interest rate is 5.00.** The first call, `myAccount.getInterest(5.00)`, refers to the `myAccount` object and to the values stored in the `myAccount` object's fields. (See Figure 7-5.) When this call is made, the expression `balance * percentageRate / 100.00` stands for 24.02 * 5.00 / 100.00.
- **In the second call, the balance is 55.63 and the interest rate is 7.00.** In the `main` method, just before this second call is made, the variable `yourInterestRate` is assigned the value `7.00`. The call itself, `yourAccount.getInterest(yourInterestRate)`, refers to the `yourAccount` object and to the values stored in the `yourAccount` object's fields. (Again, see Figure 7-5.) So, when the call is made, the expression `balance * percentageRate / 100.00` stands for 55.63 * 7.00 / 100.00.

FIGURE 7-4: Running the code in Listings 7-5 and 7-6.

```
Barry Burd (222 Cyberspace Lane) has $24.02 plus $1.2009999999999998 interest
Jane Q. Public (111 Consumer Street) has $55.63 plus $3.8941000000000003 interest
```

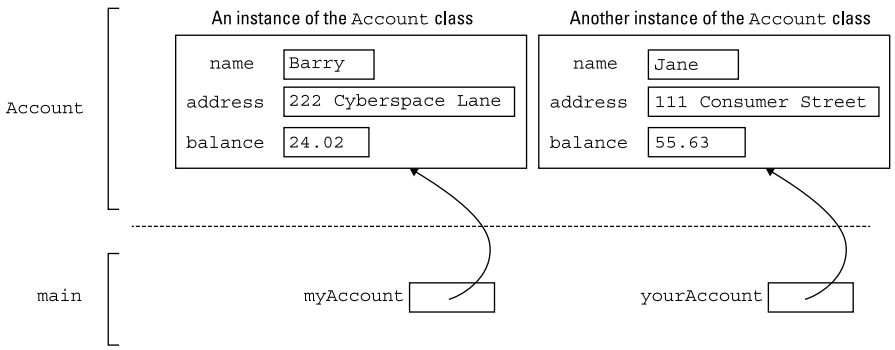


FIGURE 7-5: My account and your account.

By the way, the `main` method in Listing 7-6 contains two calls to `getInterest`. One call has the literal `5.00` in its parameter list; the other call has the variable `yourInterestRate` in its parameter list. Why does one call use a literal and the other call use a variable? No reason. I just want to show you that you can do it either way.

Passing a value to a method

Take a look at the `getInterest` method's header (as you read the explanation in the next few bullets, you can follow some of the ideas visually with the diagram in Figure 7-6):

- **The first word in the method's header (the word *double*) tells Java that when the `getInterest` method is called, the `getInterest` method returns a `double` value back to the place that called it.** The statement in the `getInterest` method's body confirms this. The statement says `return balance * percentageRate / 100.00`, and the expression `balance * percentageRate / 100.00` has type `double`. (That's because all the things in the expression — `balance`, `percentageRate`, and `100.00` — have type `double`.)

  When the `getInterest` method is called, the `return` statement calculates `balance * percentageRate / 100.00` and hands the calculation's result back to the code that called the method.

- **The word *getInterest* is the method's name.** That's the name you use to call the method when you're writing the code in Listing 7-6.

- **The parentheses contain all the things you pass to the method when you call it.** When you call a method, you can pass information to that method on the fly. This information is the method's parameter list. The `getInterest` method's header says that the `getInterest` method takes one piece of information and that piece of information must be of type `double`:

  ```
  double getInterest(double percentageRate)
  ```

  Sure enough, if you look at the first call to `getInterest` (in Listing 7-6), that call has the number `5.00` in it. And `5.00` is a `double` literal. When I call `getInterest`, I'm giving the method a value of type `double`.

  If you don't remember what a literal is, see Chapter 4.

  The same story holds true for the second call to `getInterest`. Down near the bottom of Listing 7-6, I call `getInterest` and feed the variable `yourInterestRate` to the method in its parameter list. Luckily for me, I declared `yourInterestRate` to be of type `double` just a few lines before that.

CROSS REFERENCE

When you run the code in Listings 7-5 and 7-6, the flow of action isn't from top to bottom. The action goes from `main` to `getInterest`, and then back to `main`, and then back to `getInterest`, and, finally, back to `main` again. Figure 7-7 shows the whole business.

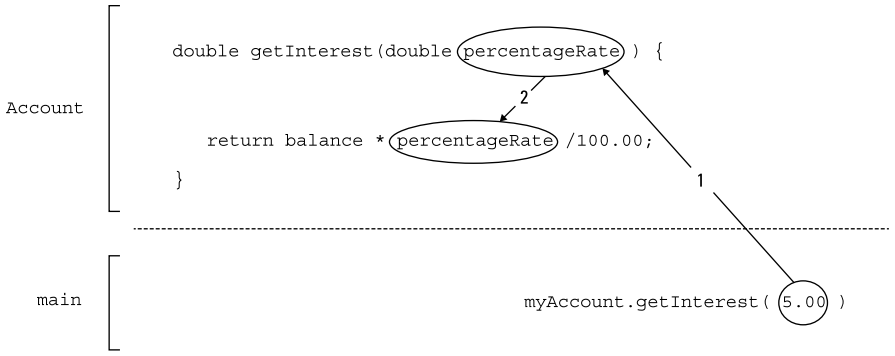


FIGURE 7-6: Passing a value to a method.

```
import java.io.IO;

class Account {
    // Yada, yada, yada,. . .

    double getInterest(double percentageRate) {
        return balance * percentageRate / 100.00;
    }
}


void main() {
    var myAccount = new Account();
    var yourAccount = new Account();

    myAccount.name = "Barry Burd";
    myAccount.address = "222 Cyberspace Lane";
    myAccount.balance = 24.02;

    yourAccount.name = "Jane Q. Public";
    yourAccount.address = "111 Consumer Street";
    yourAccount.balance = 55.63;

    myAccount.display();
    println(" plus $" + myAccount.getInterest(5.00) + " interest");

    yourAccount.display();
    double yourInterestRate = 7.00;

    double yourInterestAmount = yourAccount.getInterest(yourInterestRate);


    println(" plus $" + yourInterestAmount + " interest");
}
```


FIGURE 7-7: The flow of control in Listings 7-5 and 7-6.

Returning a value from the getInterest method

When the `getInterest` method is called, the method executes the one statement that's in the method's body: a `return` statement. The `return` statement computes

the value of `balance * percentageRate / 100.00`. If `balance` happens to be `24.02`, and `percentageRate` is `5.00`, the value of the expression is `1.201` — around $1.20. (Because the computer works exclusively with 0s and 1s, Java gets this number wrong by an ever-so-tiny amount. Java gets 1.2009999999999998. This inaccuracy is something we humans have to live with.)

After this 1.2009999999999998 value is calculated, Java executes the `return`, which sends the value back to the place in `main` where `getInterest` was called. At that point in the process, the entire method call — `myAccount.getInterest(5.00)` — takes on the value 1.2009999999999998. The call itself is inside a `println`:

```
out.println(myAccount.getInterest(5.00));
```

So the `println` ends up with the following meaning:

```
out.println(1.2009999999999998);
```

The whole process, in which a value is passed back to the method call, is illustrated in Figure 7-8.

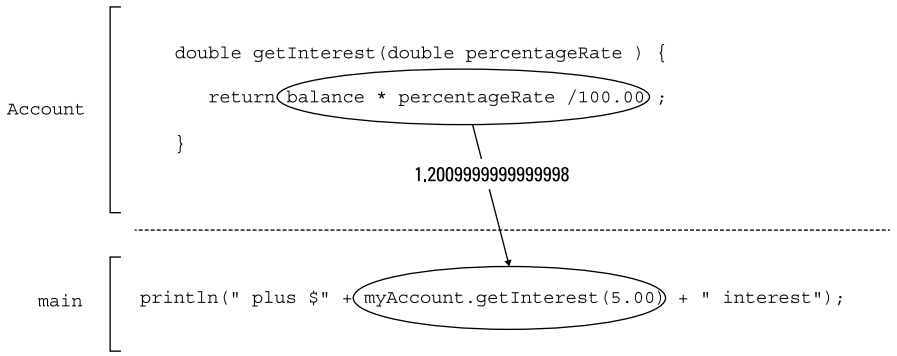


FIGURE 7-8: A method call is an expression with a value.

REMEMBER

If a method returns anything, a call to the method is an expression with a value. That value can be printed, assigned to a variable, added to something else, or whatever. Anything you can do with any other kind of value, you can do with a method call.

You might use the `Account` class in Listing 7-5 to solve a real problem. You'd call the `Account` class's `display` and `getInterest` methods in the course of an actual banking application. But the code in Listing 7-6 is artificial. That code creates some fake account data and then calls some `Account` class methods to convince you that the `Account` class's code works correctly. (You don't seriously think

that a bank has depositors named "Jane Q. Public" and "Barry Burd," do you?) The code in Listing 7-6 is a *test case* — a short-lived class whose sole purpose is to test another class's code. Like the code in Listing 7-6, each test case in this book is an ordinary main method. Ordinary main methods aren't optimal for performing tests. Java developers have something better — a more disciplined way of writing test cases. The "better way" is called JUnit, and it's described at `https://junit.org/junit5/docs/current/user-guide`.

So ends this chapter's "Sending Values to and from Methods" section. I have more to say about all this, but I don't want to put it in the next section. Instead, I put it in the "Passing Parameters" section at the end of this chapter.

TAX DAY

In previous sections, you create your own `Organization` class. Add a method to the class that computes the amount of tax the organization pays. A profit-making organization pays 10 percent of its revenue in tax, but a nonprofit organization pays only 2 percent of its revenue in tax.

Make a main method that creates two or three organizations and displays information about each organization, including the amount of tax the organization pays.

COST OF CONSUMPTION

In previous sections, you create your own `FoodProduct` class. Add methods to the class to compute the cost per 100 grams, the cost per serving, and the total number of calories in the product.

Make a main method that creates two or three products and displays information about each product.

Giving Your Numbers a Makeover

Looking at Figure 7-4 again, you may be concerned that the interest on my account is only $1.2009999999999998. Seemingly, the bank is cheating me out of two hundred-trillionths of a cent. I should head straight to the bank and demand my fair interest. Maybe you and I should go together. We'll kick up some fur at that old bank and bust this scam right open. If my guess is correct, this is part of a big salami scam. In a *salami scam,* someone shaves tiny amounts off millions of accounts. People don't notice their tiny little losses, but the person doing the shaving collects enough for a quick escape to Barbados (or for a whole truckload of salami).

Wait a minute! What about you? In Listing 7-6, you have `yourAccount`. And in Figure 7-4, your name is Jane Q. Public. Nothing is motivating you to accompany me to the bank. Checking Figure 7-4 again, I see that you're way ahead of the game. According to my calculations, the program overpays you by three hundred-trillionths of a cent. Between the two of us, we're ahead by a hundred-trillionth of a cent. What gives?

Well, because computers use 0s (zeros) and 1s and don't have an infinite amount of space to do calculations, such inaccuracies as the ones shown in Figure 7-4 are normal. The quickest solution is to display the inaccurate numbers in a more sensible fashion. You can round the numbers and display only two digits beyond the decimal point, and some handy tools from Java's API (application programming interface) can help. Listing 7-7 shows the code, and Figure 7-9 displays the pleasant result.

LISTING 7-7: **Making Your Numbers Look Right**

```
void main() {
    var yourAccount = new Account();

    yourAccount.name = "Jane Q. Public";
    yourAccount.address = "111 Consumer Street";
    yourAccount.balance = 55.63;

    yourAccount.display();
    double yourInterest = yourAccount.getInterest(7.00);
    println(" plus " + String.format("$%.2f", yourInterest) + " interest");
}
```

FIGURE 7-9: A number that looks like a dollar amount.

```
Jane Q. Public (111 Consumer Street) has $55.63 plus $3.89 interest
```

The inaccurate numbers in Figure 7-4 come from the computer's use of 0s and 1s. A mythical computer whose circuits were wired to use digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 wouldn't suffer from the same inaccuracies. So, to make things better, Java provides its own, special way around the computer's inaccurate calculations. Java's API has a class named `BigDecimal` — a class that bypasses the computer's strange 0s and 1s and uses ordinary decimal digits to perform arithmetic calculations. For more information, visit this book's website (`http://javafordummies.allmycode.com`).

Listing 7-7 uses `format` — a method belonging to the `String` class. In a call to `format`, the first parameter is a *format string* — a funny-looking code that describes exactly how the second parameter should look. To find out how format strings work, see Figure 7-10.

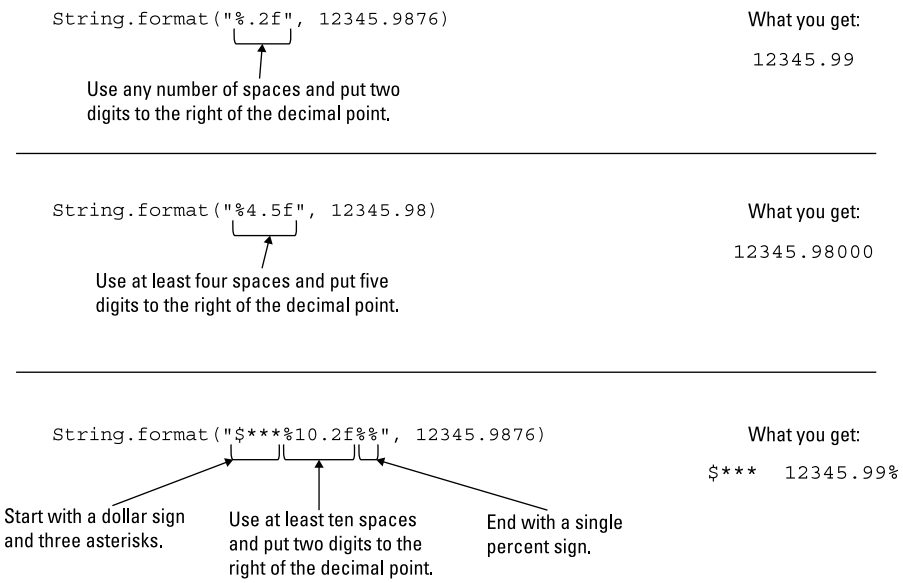

String.format("%.2f", 12345.9876)
Use any number of spaces and put two digits to the right of the decimal point.
What you get:
12345.99
String.format("%4.5f", 12345.98)
Use at least four spaces and put five digits to the right of the decimal point.
What you get:
12345.98000
String.format("$***%10.2f%%", 12345.9876)
Start with a dollar sign and three asterisks.
Use at least ten spaces and put two digits to the right of the decimal point.
End with a single percent sign.
What you get:
$*** 12345.99%

FIGURE 7-10: Using a format string.

For more examples using the `String` class's `format` method, see Chapters 8 and 9. For a complete list of options associated with the method's format string, see the `java.util.Formatter` page of Java's API documentation at `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Formatter.html`.

REMEMBER

The `String` class's `format` method doesn't change the way a number is stored internally for calculations. The only thing this `format` method does is create a nice-looking bunch of characters that can be displayed on your screen.

CROSS REFERENCE

For working with currency amounts, this section's `String.format` tricks are fairly primitive. To discover better ways of dealing with currency amounts (such as the interest amounts in this section's example), see Chapter 12.

FILL IN THE BLANKS

TRY IT OUT

Here's a Java "unprogram." It's not a real Java program because I've replaced each format string with three blank spaces:

```
void main() {
    println(String.format("   ", Math.PI));
    println(String.format("   ", Math.PI));
    println(String.format("   ", Math.PI));
    println(String.format("   ", Math.PI));
    println(String.format("   ", 7));   // Hint: Use d instead of f
    println(String.format("   ", -7));
    println(String.format("   ", +7));
    println(String.format("   ", +7));
    println(String.format("   ", -7));
    println(String.format("   ", "Java"));  // Hint: Use s instead of f
    println(String.format("   ", "Java"));
```

Fill in the missing format strings so that this program produces the following output:

```
3.141593
3.142
PI = 3.141593
         3
    7
   -7
    7
   +7
-0007
Java
JAVA
```

To do this, look for clues in the `java.util.Formatter` page of Java's API documentation at `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Formatter.html`.

Hide-and-Seek

Put down this book and put on your hat. You've been such a loyal reader that I'm taking you out to lunch!

I have just one problem. I'm a bit short on cash. Would you mind if, on the way to lunch, we stopped at an automatic teller machine and picked up a few bucks? Also, we have to use your account. My account is a little low.

Fortunately, the teller machine is easy to use. Just step right up and enter your PIN. After you enter your PIN, the machine asks which of several variable names you want to use for your current balance. You have a choice of `balance324`, `myBal`, `currentBalance`, `b$`, `BALANCE`, `asj999`, or `conStanTinople`. Having selected a variable name, you're ready to select a memory location for the variable's value. You can select any number between 022FFF and 0555AA. (Those numbers are in hexadecimal format.) After you configure the teller machine's software, you can easily get your cash. You did bring a screwdriver, didn't you?

Good programming

When it comes to good computer programming practice, one word stands out above all others: *simplicity.* When you're writing complicated code, the last thing you want is to deal with somebody else's misnamed variables, convoluted solutions to problems, or clever, last-minute kludges. You want a clean interface that makes you solve your own problems and no one else's.

In the automatic teller machine scenario that I describe earlier, the big problem is that the machine's design forces you to worry about other people's concerns. When you should be thinking about withdrawing money for lunch, you're thinking instead about variables and storage locations. Sure, someone has to work out the teller machine's engineering problems, but the banking customer isn't the person.

REMEMBER

This section is about safety, not security. Safe code keeps you from making accidental programming errors. Secure code (a completely different story) keeps malicious hackers from doing intentional damage.

So, everything connected with every aspect of a computer program has to be simple, right? Well, no. That's not right. Sometimes, to make things simple in the long run, you have to do lots of preparatory work upfront. The people who built the automated teller machine worked hard to make sure that the machine is consumer-proof. The machine's interface, with its screen messages and buttons, makes the machine a complicated, but carefully designed, device.

The point is that making things look simple takes some planning. In the case of object-oriented programming, one way to make things look simple is to prevent code outside a class from directly using fields defined inside the class. Take a peek

at the code in Listing 7-1. You're working at a company that has just spent $10 million for the code in the `Account` class. (That's $2 million per line!) Now your job is to write the main method. You would like to write

```
myAccount.name = "Barry Burd";
```

but doing so would be submerging you too far inside the guts of the `Account` class. After all, people who use an automatic teller machine aren't allowed to program the machine's variables. They can't use the machine's keypad to type the statement

```
balanceOnAccount29872865457 = balanceOnAccount29872865457 + 1000000.00;
```

Instead, they push buttons that do the job in an orderly manner. That's how a programmer achieves safety and simplicity.

To keep things nice and orderly, you need to change the `Account` class from Listing 7-1 by outlawing such statements as

```
myAccount.name = "Barry Burd";
```

and

```
print(yourAccount.balance);
```

Of course, this poses a problem. You're the person who's writing the code for the main method. If you can't write `myAccount.name` or `yourAccount.balance`, how will you accomplish anything at all? The answer lies in things called *accessor methods.* Listings 7-8 and 7-9 demonstrate these methods.

LISTING 7-8: **Hide Those Fields**

```
class Account {
    private String name;
    private String address;
    private double balance;

    public void setName(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    public void setAddress(String a) {
        address = a;
    }
```

```
    public String getAddress() {
        return address;
    }
    public void setBalance(double b) {
        balance = b;
    }
    public double getBalance() {
        return balance;
    }
}
```

LISTING 7-9: Calling Accessor Methods

```
void main() {
    var myAccount = new Account();
    var yourAccount = new Account();

    myAccount.setName("Barry Burd");
    myAccount.setAddress("222 Cyberspace Lane");
    myAccount.setBalance(24.02);

    yourAccount.setName("Jane Q. Public");
    yourAccount.setAddress("111 Consumer Street");
    yourAccount.setBalance(55.63);

    println(myAccount.getName() + " (" + myAccount.getAddress() + ") " +
            "has $" + myAccount.getBalance());
    println(yourAccount.getName() + " (" + yourAccount.getAddress() + ") " +
            "has $" + yourAccount.getBalance());
}
```

A run of the code in Listings 7-8 and 7-9 looks no different from a run of Listings 7-1 and 7-2. Either program's run is shown earlier, in Figure 7-3. The big difference is that in Listing 7-8, the `Account` class enforces the carefully controlled use of its `name`, `address`, and `balance` fields.

Public lives and private dreams: Making a field inaccessible

Notice the addition of the word *private* in front of each of the `Account` class's field declarations. The word *private* is a Java keyword. When a field is declared private, no code outside of the class can make direct reference to that field. So, if you put `myAccount.name = "Barry Burd"` in Listing 7-9, you get an error message such as `name has private access in Account`.

Rather than reference `myAccount.name`, the Listing 7-9 programmer must call method `myAccount.setName` or method `myAccount.getName`. These methods, `setName` and `getName`, are called *accessor* methods because they provide access to the `Account` class's `name` field. (Actually, the term *accessor method* isn't formally a part of the Java programming language. It's just the term that people use for methods that do this sort of thing.) To zoom in even more, `setName` is called a *setter* method, and `getName` is called a *getter* method. (I bet you won't forget that terminology!)

TIP

With many IDEs, you don't have to type your own accessor methods. First, you type a field declaration like `private String name`. Then, on your IDE's menu bar, you choose Source➪Generate Getters and Setters or choose Code➪Insert Code➪Setter or some mix of those commands. After you make all your choices, the IDE creates accessor methods and adds them to your code.

Notice that all the setter and getter methods in Listing 7-8 are declared to be `public`. This ensures that anyone from anywhere can call these two methods. The idea here is that manipulating the actual fields from outside the `Account` code is impossible, but you can easily reach the approved setter and getter methods for using those fields.

Think again about the automatic teller machine. Someone using the ATM can't type a command that directly changes the value in their account's `balance` field, but the procedure for depositing a million-dollar check is easy to follow. The people who build the teller machines know that if the check-depositing procedure is complicated, plenty of customers will mess it up royally. So that's the story — make impossible anything that people shouldn't do and make sure that the tasks people should be doing are easy.

TIP

Nothing about having setter and getter methods is sacred. You don't have to write a particular setter or getter method if no one will ever call it. In some situations, you have fields that you don't want anyone else's code to access. So, for those fields, you don't create setters and getters.

TIP

When you create a method to set the value in a `balance` field, you don't have to name your method `setBalance`. You can name it `tunaFish` or whatever you like. The trouble is that the `set`*`Fieldname`* convention (with lowercase letters in `set` and an uppercase letter to start the *`Fieldname`* part) is an established stylistic convention in the world of Java programming. If you don't follow the convention, you confuse the kumquats out of other Java programmers.

REMEMBER

When you call a setter method, you feed it a value of the type that's being set. That's why, in Listing 7-9, you call `yourAccount.setBalance(55.63)` with a parameter of type `double`. In contrast, when you call a getter method, you usually don't feed any values to the method. That's why, in Listing 7-9, you call

`yourAccount.getBalance()` with an empty parameter list. Occasionally, you may want to get and set a value with a single statement. To add a dollar to your account's existing balance, you write `yourAccount.setBalance(yourAccount.getBalance() + 1.00)`.

Enforcing rules with accessor methods

Go back to Listing 7-8 and take a quick look at the `setName` method. Imagine putting the method's assignment statement inside an `if` statement:

```
public void setName(String n) {
    if (!n.equals("")) {
        name = n;
    }
}
```

Now, if the programmer in charge of Listing 7-9 writes `myAccount.setName("")`, the call to `setName` has no effect. Furthermore, because the `name` field is private, the following statement is illegal in Listing 7-9:

```
myAccount.name = "";
```

Of course, a call such as `myAccount.setName("Joe Schmoe")` still works because `"Joe Schmoe"` doesn't equal the empty string `""`.

That's cool. With a private field and an accessor method, you can prevent someone from assigning the empty string to an account's `name` field. With more elaborate `if` statements, you can enforce any rules you want.

AVOIDING PRIVATE AND PUBLIC ACCESS

One way to create a field or method that's between `private` and `public` is to use neither of these words. This is called *default access*. It's what you see in all the listings that come before this main "Hide-and-Seek" section. For example, Listing 7-1 uses default access because it begins with

```
class Account {
    String name;
    String address;
```

(continued)

(continued)

and not with

```
class Account {
    public String name;
    private String address;
```

Professional programmers shun the use of default access because default access doesn't shield a field from accidental misuse. But in my experience, you learn best when you learn about the simplest stuff first. And in Java, default access is the simplest stuff. In this book, I delay the discussion of private access until this chapter's "Hide-and-Seek" section, and I delay a detailed discussion of access modifiers until Chapter 11. As you read this book's earlier examples, please keep in mind that default access isn't the best thing to use in a Java program. And, if a professional programmer asks you where you learned to use default access, please lie and blame someone else's book.

PRIVATE EYE

TRY IT OUT

In previous sections, you create your own `Organization` and `FoodProduct` classes. In those classes, replace the default access fields with private fields. Create getter and setter methods for those fields. In the setter methods, add code to ensure that the `String` values aren't empty and that numeric values aren't negative.

Streamlining Java

Every method belongs inside a class. Just think about the method calls in this book's listings. Java's `format` method lives inside the `String` class.

```
println(" plus " + String.format("$%.2f", yourInterest) + " interest");
```

The `parseInt` method is part of Java's `Integer` class.

```
int inputNumber = Integer.parseInt(input);
```

The `display` and `getInterest` methods in Listing 7-5 sit comfortably inside an `Account` class.

```
class Account {
    // ...
```

```
    void display() {
    // ...

    double getInterest(double percentageRate) {
    // ...
}
```

So, what about all the `main` methods in Chapters 1–6? What classes do `main` methods belong to?

If you create a `main` method and you don't explicitly put that `main` method inside a class, Java does some extra work for you. With no fanfare at all, Java declares a class to contain your `main` method. In other words, Java creates an *implicit class* (or, as the experts like to say, an *implicitly declared class*). A file with an implicit class is called a *simple source file*.

Imagine a file named `UseAccount.java` containing the code in Listing 7-7. When you look at the file in an IDE or a text editor, you see only the code in Listing 7-7. But, when Java runs the code, Java pretends to see the code in Listing 7-10.

LISTING 7-10: Java Sees What It Wants to See

```
class UseAccount {

    void main() {
        var yourAccount = new Account();

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        yourAccount.display();
        double yourInterest = yourAccount.getInterest(7.00);
        java.io.IO.println(" plus " +
                  String.format("$%.2f", yourInterest) + " interest");
    }
}
```

In earlier sections, I likened a class to the header row of a table. You may ask whether that analogy fits for the class in Listing 7-10. The answer may not be very satisfying, but I've mapped it out in Table 7-3.

TABLE 7-3 The UseAccount Class, Pictured as a Table

Main
(All the code in the `main` method's body)

The class in Table 7-2 and Listing 7-3 has four members: `name`, `address`, `balance`, and `display`. In contrast, the class in Table 7-3 and Listing 7-10 has only one member: `main`. Table 7-3 isn't much, but it's all we've got.

Until recently, Java didn't have simple source files and didn't create implicit classes for you. Instead, to create a `main` method, you'd have to write all the code in Listing 7-11.

LISTING 7-11: Out With the New; In With the Old

```
public class UseAccount {

    public static void main(String[] args) {
        var yourAccount = new Account();

        yourAccount.name = "Jane Q. Public";
        yourAccount.address = "111 Consumer Street";
        yourAccount.balance = 55.63;

        yourAccount.display();
        double yourInterest = yourAccount.getInterest(7.00);
        System.out.println(" plus " +
                   String.format("$%.2f", yourInterest) + " interest");
    }
}
```

Typing the extra words in Listing 7-11 wasn't difficult. In fact, most IDEs can paste these words into the code on your behalf. But terms like `public`, `class`, `static`, and `args` camouflage the essential purpose of a program's `main` method. So, starting with Java 21, the stewards of Java chipped away at all the boilerplate terms. The result was `main` methods like the ones in previous sections.

Of course, there's no free lunch. The implicitly declared classes that surround `main` methods have some disadvantages. But the disadvantages are esoteric. They arise only in special situations. In most situations, the simple `void main()` style is all you need.

In the next several chapters, I strike up a compromise. Some examples use the traditional verbose style with a `main` method inside an explicitly declared class. Other examples have the new simple source files.

The `print`, `println`, and `readln` methods that populate listings in Chapters 3–6 didn't become part of Java until September 2024, and then only as a preview feature. Until then, the easiest way to display text on the screen was to call the `System.out.println` method. (See Listing 7-11.)

PUBLIC CLASSES

The first line of Listing 7-11 is

```
public class UseAccount {
```

The `UseAccount` class is public. A public class is available for use by all other classes. For example, if you write an `ATMController` program in some remote corner of cyberspace, then your `ATMController` program can call on the code in the `UseAccount` class of Listing 7-11. (Of course, your code has to know where in cyberspace I've stored the code in Listing 7-11, but that's another story.)

Declaring some of my classes to be public makes me feel good. (Yes, programmers do certain things to feel good.) The aesthetic sense of goodness comes from the fact that my class may be useful to other programmers. When I create a class that declares something useful and nameable — an `Account`, an `Engine`, a `Customer`, a `BrainWave`, a `Headache`, or a `SevenLayerCake` class — I declare the class to be public.

When a class contains a `main` method, Java programmers tend to make the class public without thinking much about who uses the class. So, even if no other class makes use of my `main` method, I declare the class in Listing 7-11 to be public.

When you declare a class to be public, you must declare the class in a file whose name is exactly the same as the name of the class (but with the `.java` extension added). For example, if you declare `public class MyImportantCode`, you must put the class's code in a file named `MyImportantCode.java`, with uppercase letters `M`, `I`, and `C` and all other letters lowercase. This file-naming rule has an important consequence: If your code declares two public classes, your code must consist of at least two `.java` files. In other words, you can't declare two `public` classes in one `.java` file.

For more news about the word *public* and other such words, see Chapter 11.

PLAYING THE LONG GAME

Rewrite the first listing in Chapter 5 (this book's first Guessing Game example) in the style of Listing 7-11. (If you can, test your code in an environment that runs Java 20 or earlier.)

Experiments with Variables

One summer during my college days, I was sitting on the front porch, loafing around and talking with someone I'd just met. I think her name was Janine. "Where are you from?" I asked. "Mars," she answered. She paused to see whether I'd ask a follow-up question.

As it turned out, Janine was from Mars, Pennsylvania, a small town about 20 miles north of Pittsburgh. Okay, what's my point? The point is that the meaning of a name depends on the context. If you're just north of Pittsburgh and ask, "How do I get to Mars from here?" you may receive a sensible, nonchalant answer. But if you ask the same question standing on a street corner in Manhattan, you'll probably arouse some suspicion. (Okay, knowing Manhattan, people would probably just ignore you.)

Of course, the people who live in Mars, Pennsylvania, are very much aware that their town has an oddball name. Fond memories of teenage years at Mars High School don't prevent a person from knowing about the big red planet. On a clear evening in August, you can still have the following conversation with one of the local residents:

- *You:* How do I get to Mars?
- *Local resident:* You're in Mars, pal. What particular part of Mars are you looking for?
- *You:* No, I don't mean Mars, Pennsylvania. I mean the planet Mars.
- *Local resident:* Oh, the planet! Well, then, catch the 8:19 train leaving for Cape Canaveral. No, wait — that's the local train. That'd take you through West Virginia . . . .

The meaning of a name depends on where you're using the name. Although most English-speaking people think of Mars as a place with a carbon dioxide atmosphere, some folks in Pennsylvania think about all the shopping they can do in Mars. And those folks in Pennsylvania really have two meanings for the name *Mars.* In Java, those names may look like this: `Mars` and `planets.Mars`.

Putting a variable in its place

Your first experiment is shown in Listings 7-12 and 7-13. The listings' code highlights the difference between two kinds of variables:

- **Field:** A variable that's *not* declared inside of a method.
- **Method-local variable:** A variable that *is* declared inside of a method.

LISTING 7-12: Two Meanings for Mars

```
import java.io.IO;

class EnglishSpeakingWorld {
    String mars = " red planet";              // field

    void visitPennsylvania() {
        IO.println("visitPA is running:");

        String mars = " Janine's home town";   // method-local variable

        IO.println(mars);         // Janine's home town
        IO.println(this.mars);    // red planet
    }
}
```

LISTING 7-13: Calling the Code of Listing 7-12

```
void main() {
    println("main is running:");
    var e = new EnglishSpeakingWorld();

    //out.println(mars); cannot resolve symbol
    println(e.mars);
    e.visitPennsylvania();
}
```

Figure 7-11 shows a run of the code in Listings 7-12 and 7-13. Figure 7-12 shows a diagram of the code's structure. Notice that, in this example, I choose to put the `main` method *outside* of the `EnglishSpeakingWorld` class.

The `main` method creates an instance of the `EnglishSpeakingWorld` class. The variable e refers to the new instance. The new instance is an object with a variable named `mars` inside it. That `mars` variable has the value `"red planet"`. This `mars` variable is a field.

```
main is running:
   red planet
visitPA is running:
   Janine's home town
   red planet
```

FIGURE 7-11: A run of the code in Listings 7-12 and 7-13.

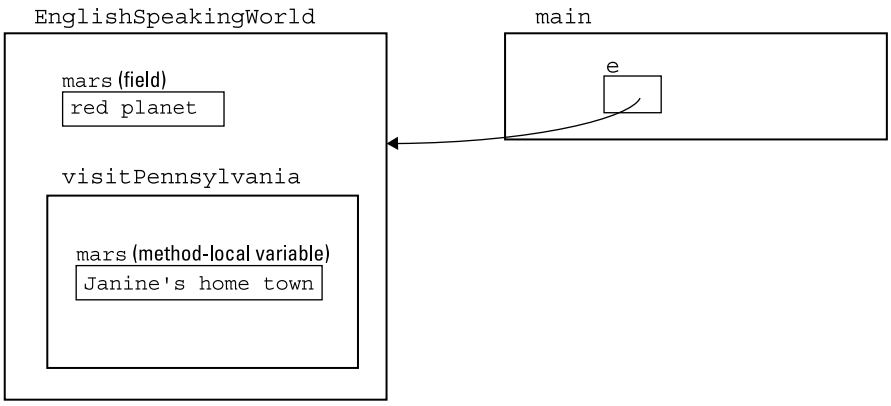


FIGURE 7-12: The structure of the code in Listings 7-12 and 7-13.

TECHNICAL STUFF

Another way to describe that `mars` field is to call it an *instance variable* because that `mars` variable (the variable whose value is `"red planet"`) belongs to an *instance* of the `EnglishSpeakingWorld` class.

Now look at the `main` method in Listing 7-13. Inside the `main` method, you aren't permitted to write `println(mars)`. In other words, a bare-faced reference to any `mars` variable is a definite no-no. The `mars` variable that I mention in the preceding paragraph belongs to the `EnglishSpeakingWorld` class. But the `main` method lives outside of the `EnglishSpeakingWorldClass`. Too bad for that!

However, inside the main method, you can certainly write `e.mars` because the e variable refers to your `EnglishSpeakingWorld` object. That's nice.

Near the bottom of Listing 7-13, you call the `visitPennsylvania` method. When you're inside `visitPennsylvania`, you have another declaration of a `mars` variable, whose value is `"Janine's home town"`. This particular `mars` variable is a method-local variable because it belongs to just one method: the `visitPennsylvania` method.

Now you have to worry about two variables in Listing 7-12, both with the name *mars.* One `mars` variable, a field, has the value `"red planet"`. The other `mars` variable, a method-local variable, has the value `"Janine's home town"`. In Listing 7-12, when you use the word *mars,* which of the two variables are you referring to?

The answer is, when you're visiting Pennsylvania, the variable with value `"Janine's home town"` wins. When in Pennsylvania, think the way the Pennsylvanians think. When you're executing code inside the `visitPennsylvania` method, resolve any variable name conflicts by going with method-local variables — variables declared right inside the `visitPennsylvania` method.

What if you're in Pennsylvania and need to refer to that 2-mooned celestial object? More precisely, how does code inside the `visitPennsylvania` method refer to the field with value `"red planet"`? The answer is, use `this.mars`. The word *this* points to whatever object contains all this code (and not to any methods inside the code). That object, an instance of the `EnglishSpeakingWorld` class, has a big, fat `mars` field, and that field's value is `"red planet"`. So that's how you can force code to see outside the method it's in — you use the Java keyword `this`.

CROSS REFERENCE

For more information on the keyword `this`, see Chapter 10.

Telling a variable where to go

Years ago, when I lived in Milwaukee, Wisconsin, I made frequent use of the local bank's automatic teller machines. Machines of this kind were just beginning to become standardized. The local teller machine system was named TYME, which stood for Take Your Money Everywhere.

I remember traveling by car out to California. At one point, I got hungry and stopped for a meal, but I was out of cash. So I asked a gas station attendant, "Do you know where there's a TYME machine around here?"

So you see, a name that works well in one place could work terribly, or not at all, in another place. In Listings 7-14 and 7-15, I illustrate this point (with more than just an anecdote about teller machines).

LISTING 7-14: **Tale of Atomic City**

```
import java.io.IO;

class EnglishSpeakingWorld2 {
    String mars;
```

(continued)

LISTING 7-14: ***(continued)***

```
    void visitIdaho() {
        IO.println("visitID is running:");

        mars = " red planet";
        String atomicCity = " Population: 25";

        IO.println(mars);
        IO.println(atomicCity);
    }

    void visitNewJersey() {
        IO.println("visitNJ is running:");
        IO.println(mars);
        // IO.println(atomicCity); cannot resolve symbol
    }
}
```

LISTING 7-15: **Calling the Code of Listing 7-14**

```
void main() {
    var e = new EnglishSpeakingWorld2();

    e.visitIdaho();
    e.visitNewJersey();
}
```

Figure 7-13 shows a run of the code in Listings 7-14 and 7-15. Figure 7-14 shows a diagram of the code's structure. The code for `EnglishSpeakingWorld2` has two variables: The `mars` variable, which isn't declared inside a method, is a field; the other variable, `atomicCity`, is a method-local variable and is declared inside the `visitIdaho` method.

```
visitID is running:
   red planet
   Population: 25
visitNJ is running:
   red planet
```

FIGURE 7-13: A run of the code in Listings 7-14 and 7-15.

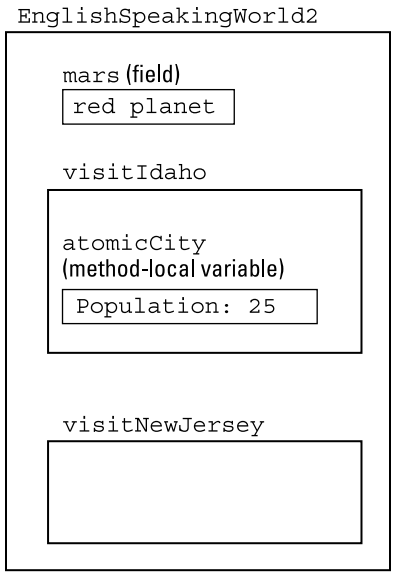


FIGURE 7-14: The structure of the code in Listings 7-14 and 7-15.

In Listing 7-14, notice where each variable can and can't be used. When you try to use the `atomicCity` variable inside the `visitNewJersey` method, you see an error message. Literally, the message says `cannot resolve symbol`. Figuratively, the message says, "Hey, buddy, Atomic City is in Idaho, not New Jersey." Technically, the message says that the method-local variable `atomicCity` is available only in the `visitIdaho` method because that's where the variable was declared.

Back inside the `visitIdaho` method, you're free to use the `atomicCity` variable as much as you want. After all, the `atomicCity` variable is declared inside the `visitIdaho` method.

And what about Mars? Have you forgotten about your old friend, that lovely 80-degrees-below-0 planet? Well, both the `visitIdaho` and `visitNewJersey` methods can access the `mars` variable. That's because the `mars` variable is a field. That is, the `mars` variable is declared in the code for the `EnglishSpeakingWorld2` class but not inside any particular method. (In my stories about the names for things, remember that people who live in both states, Idaho and New Jersey, have heard of the planet Mars.)

The life cycle of the `mars` field has three separate steps:

1. When the `EnglishSpeakingWorld2` class first flashes into existence, the computer sees `String mars` and creates space for the `mars` field.

2. When the `visitIdaho` method is executed, the method assigns the value `"red planet"` to the mars field. (The `visitIdaho` method also prints the value of the `mars` field.)
3. When the `visitNewJersey` method is executed, the method prints the mars value once again.

In this way, the `mars` field's value is passed from one method to another.

TRY IT OUT

Try out these programs. See what you think.

WHO'S WHO?

What's the output of the following code? Why?

```
import java.io.IO;

public class Main1 {
    String name = "Nancy";

    void main() {
        IO.println(name);
        String name = "Barry";
        IO.println(name);
    }
}
```

WHO ELSE IS WHO?

What's the output of the following code? Why?

```
import java.io.IO;

public class Main2 {
    String name = "George";

    void main() {
        doStuff();
    }

    void doStuff() {
        IO.println(name);
```

```
        String name = "Barry";
        IO.println(name);

        IO.println(this.name);
    }
}
```

Passing Parameters

A method can communicate with another part of your Java program in several ways. One way is by way of the method's parameter list. Using a parameter list, you pass on-the-fly information to a method as the method is being called.

Imagine that the information you pass to the method is stored in one of your program's variables. What, if anything, does the method actually do with that variable? The following sections present a few interesting case studies.

Pass by value

According to my web research, the town of Smackover, Arkansas, has 2,232 people in it. But my research isn't current. Just yesterday, Dora Kermongoos celebrated a joyous occasion over at Smackover General Hospital — the birth of her healthy, brown-eyed baby girl. (The girl weighs 7 pounds, 4 ounces, and is 21 inches tall.) Now the town's population has risen to 2,233.

Listing 7-16 has a very bad program in it. The program is supposed to add 1 to a variable that stores Smackover's population, but the program doesn't work. Take a look at Listing 7-16 to see why.

LISTING 7-16: This Program Doesn't Work

```
void main() {
    int smackoverARpop = 2232;

    birth(smackoverARpop);
    println(smackoverARpop);
}

void birth(int cityPop) {
    cityPop++;
}
```

When you run the program in Listing 7-16, the program displays the number 2232 onscreen. After nine months of planning and anticipation and Dora's whopping seven hours in labor, the Kermongoos family's baby girl wasn't registered in the system. What a shame!

The improper use of parameter passing caused the problem. In Java, when you pass a parameter that has one of the eight primitive types, that parameter is *passed by value.*

CROSS REFERENCE

For a review of Java's eight primitive types, see Chapter 4.

Here's what this means in plain English: Any changes that the method makes to the value of its parameter don't affect the values of variables back in the calling code. In Listing 7-16, the `birth` method can apply the `++` operator to `cityPop` all it wants — the application of `++` to the `cityPop` parameter has absolutely no effect on the value of the `smackoverARpop` variable back in the main method.

Technically, what's happening is the copying of a value. (See Figure 7-15.) When the `main` method calls the `birth` method, the value stored in `smackoverARpop` is copied to another memory location — a location reserved for the `cityPop` parameter's value. During the `birth` method's execution, 1 is added to the `cityPop` parameter. But the place where the original `2232` value was stored — the memory location for the `smackoverARpop` variable — remains unaffected.

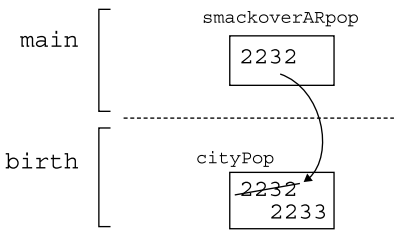


FIGURE 7-15: Pass by value, under the hood.

REMEMBER

When you do parameter passing with any of the eight primitive types, the computer uses *pass by value.* The value stored in the calling code's variable remains unchanged. This happens even if the calling code's variable and the called method's parameter happen to have exactly the same name.

Returning a result

You must fix the problem that the code in Listing 7-16 poses. After all, a young baby Kermongoos can't go through life untracked. To record this baby's existence,

you have to add 1 to the value of the `smackoverARpop` variable. You can do this in plenty of ways, and the way presented in Listing 7-17 isn't the simplest. Even so, the way shown in Listing 7-17 illustrates a point: Returning a value from a method call can be an acceptable alternative to parameter passing. Look at Listing 7-17 to see what I mean.

LISTING 7-17: This Program Works

```
void main() {
    int smackoverARpop = 2232;

    smackoverARpop = birth(smackoverARpop);
    println(smackoverARpop);
}

int birth(int cityPop) {
    return cityPop + 1;
}
```

After running the code in Listing 7-17, the number you see on your computer screen is the correct number: 2233.

The code in Listing 7-17 has no new features in it (unless you call *working correctly* a new feature). The most important idea in Listing 7-17 is the `return` statement, which also appears in Chapter 7. Even so, Listing 7-17 presents a nice contrast to the approach in Listing 7-16, which had to be discarded.

Pass by reference

In the previous section or two, I take great pains to emphasize a certain point — that when a parameter has one of the eight primitive types, the parameter is passed by value. If you read this, you probably missed the emphasis on the parameter's having one of the eight primitive types. The emphasis is needed because passing objects (reference types) doesn't quite work the same way.

When you pass an object to a method, the object is *passed by reference.* What this means to you is that statements in the called method *can* change any values that are stored in the object's variables. Those changes *do* affect the values that are seen by whatever code called the method. Listings 7-18 and 7-19 illustrate the point.

LISTING 7-18: What Is a City?

```
class City {
    int population;
}
```

LISTING 7-19: Passing an Object to a Method

```
void main() {
    var smackoverAR = new City();
    smackoverAR.population = 2232;
    birth(smackoverAR);
    println(smackoverAR.population);
}

void birth(City aCity) {
    aCity.population++;
}
```

When you run the code in Listings 7-18 and 7-19, the output you see is the number 2233. That's good because the code has things like `++` and the word *birth* in it. The deal is, adding 1 to `aCity.population` inside the `birth` method actually changes the value of `smackoverAR.population`, as it's known in the `main` method.

To see how the `birth` method changes the value of `smackoverAR.population`, look at Figure 7-16. When you pass an object to a method, the computer doesn't make a copy of the entire object. Instead, the computer makes a copy of a *reference to* that object. (Think of it the way it's shown in Figure 7-16. The computer makes a copy of an arrow that points to the object.)

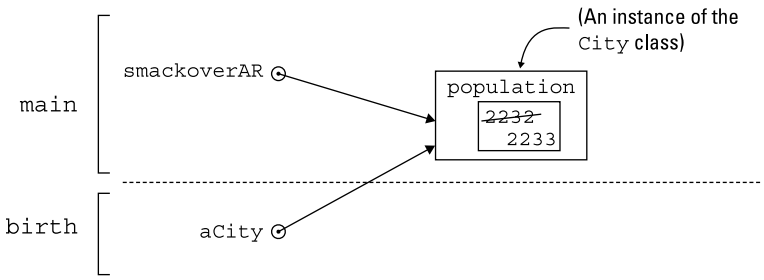


FIGURE 7-16: Pass by reference, under the hood.

In Figure 7-16, you see just one instance of the `City` class, with a `population` variable inside it. Now keep your eye on that object as you read the following steps:

1. Just before the `birth` method is called, the `smackoverAR` variable refers to that object — the instance of the `City` class.
2. When the `birth` method is called and `smackoverAR` is passed to the `birth` method's `aCity` parameter, the computer copies the reference from `smackoverAR` to `aCity`. Now `aCity` refers to that same object — the instance of the `City` class.
3. When the statement `aCity.population++` is executed inside the `birth` method, the computer adds 1 to the object's `population` field. Now the program's one-and-only `City` instance has 2233 stored in its `population` field.
4. The flow of execution goes back to the `main` method. The value of `smackoverAR.population` is printed. But `smackoverAR` refers to that one instance of the `City` class. So `smackoverAR.population` has the value 2233. The Kermongoos family is so proud.

Returning an object from a method

Believe it or not, the previous sections on parameter passing left unexplored one nook-and-cranny of Java methods: When you call a method, the method can return something right back to the calling code. In previous chapters and sections, I return primitive values, such as `int` values, or nothing (otherwise known as *void*). In this section, I return a whole object. It's an object of type `City` from Listing 7-18. The code that makes this happen is in Listing 7-20.

LISTING 7-20: Here, Have a City

```
void main() {
    var smackoverAR = new City();
    smackoverAR.population = 2232;
    smackoverAR = doBirth(smackoverAR);
    println(smackoverAR.population);
}

City doBirth(City aCity) {
    var myCity = new City();
    myCity.population = aCity.population + 1;
    return myCity;
}
```

If you run the code in Listing 7-20, you get the number 2233. That's good. The code works by telling the `doBirth` method to create another `City` instance. In the new instance, the value of `population` is `2233`. (See Figure 7-17.)

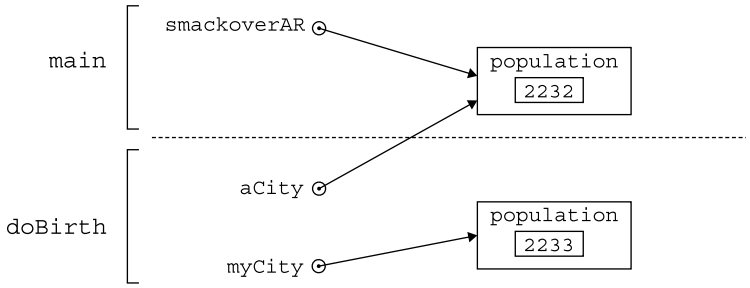


FIGURE 7-17: The `doBirth` method creates a `City` instance.

After the `doBirth` method is executed, that `City` instance is returned to the `main` method. Then, back in the `main` method, that instance (the one that `doBirth` returns) is assigned to the `smackoverAR` variable. (See Figure 7-18.) Now `smackoverAR` refers to a brand-new `City` instance — an instance whose population is 2,233.

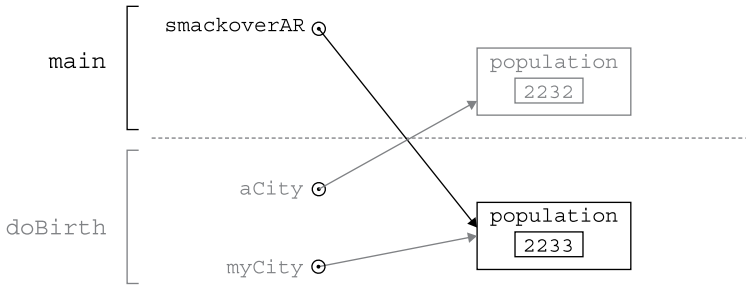


FIGURE 7-18: The new `City` instance is assigned to the `smackoverAR` variable.

In Listing 7-20, notice the type consistency in the calling and returning of the `doBirth` method:

- The `smackoverAR` variable has type `City`. The `smackoverAR` variable is passed to the `aCity` parameter, which is also of type `City`.
- The `myCity` variable is of type `City`. The `myCity` variable is sent back in the `doBirth` method's `return` statement. That's consistent, because the `doBirth` method's header begins with `static City doBirth(blah, blah, blah` . . . — a promise to return an object of type `City`.

» The `doBirth` method returns an object of type `City`. Back in the `main` method, the object that the call to `doBirth` returns is assigned to the `smackoverAR` variable, and (you guessed it) the `smackoverAR` variable is of type `City`.

Aside from being quite harmonious, all this type agreement is absolutely necessary. If you write a program in which your types don't agree with one another, the compiler spits out an unsympathetic `incompatible types` message.

A final word

Dora Kermongoos and her newborn baby daughter are resting comfortably in their Smackover, Arkansas, home. Be sure to watch the upcoming reality television series, *Smackover Newborns: Secrets from Behind Closed Doors*.

IN THIS CHAPTER

- Defining constructors
- Using Java's default constructor
- Creating a class with one line of code

Chapter 8
Constructing New Objects

Ms. Jennie Burd

121 Schoolhouse Lane

Anywhere, Kansas

Dear Ms. Burd,

In response to your letter of June 21, I believe I can say with complete assurance that objects are not created spontaneously from nothing. Although I've never actually seen an object being created (and no one else in this office can claim to have seen an object in its moment of creation), I have every confidence that some process or another is responsible for the building of these interesting and useful thingamajigs. We here at ObjectsAndClasses.com support the unanimous opinions of both the scientific community and the private sector in matters of this nature. Furthermore, we agree with the recent finding of a blue-ribbon panel that concludes, beyond any doubt, that spontaneous object creation would impede the present economic outlook.

Please be assured that I have taken all steps necessary to ensure the safety and well-being of you, our loyal customer. If you have any further questions, please do not hesitate to contact our Complaint department. The department's manager is Mr. Blake Wholl. You can contact him by visiting our company's website.

Once again, let me thank you for your concern, and I hope you continue to patronize ObjectsAndClasses.com.

Yours truly,

Mr. Scott Brickenchicker (the one who couldn't get on the elevator in Chapter 4)

Defining Constructors (What It Means to Be a Temperature)

Here's a statement that creates an object:

```
Account myAccount = new Account();
```

I know it works — I got it from one of my own examples in Chapter 7. Anyway, Chapter 7 has me saying, "When Java performs the initialization `myAccount = new Account()`, Java creates a new object (a new instance of the `Account` class) and makes the variable `myAccount` refer to that new object." What does this pithy sentence mean?

Well, the keyword `new` tells Java to create an object — an instance of a class. Java responds by performing certain actions. For starters, Java finds a place in its memory to store information about the new object. If the object has fields, the fields should eventually have meaningful values.

CROSS REFERENCE

To find out about fields, see Chapter 7.

When you ask Java to create a new object, you may want to specify what's placed in the object's fields. And what if you're interested in doing more than filling fields? Perhaps, when Java creates a new object, you have a whole list of jobs for Java to carry out. For instance, when Java creates a new window object, you want Java to realign the sizes of all buttons in that window.

Creating a new object can involve all kinds of tasks, so in this chapter you create constructors. A *constructor* tells Java to perform a new object's start-up tasks.

What is a temperature?

"Good morning, and welcome to Object News. The local temperature in your area is a pleasant 73 degrees Fahrenheit."

Each temperature consists of two parts: a number and a temperature scale. A number is just a `double` value, such as 32.0 or 70.52. But what's a temperature scale? Is it a string of characters, like `"Fahrenheit"` or `"Celsius"`? Not really, because some strings aren't temperature scales. There's no `"Quelploof"` temperature scale, and a program that can display the temperature `"73 degrees Quelploof"` is a bad program. So how can you limit the temperature scales to the small number of scales that people use? One way to do it is with Java's `enum` type.

What is a temperature scale? (Java's enum type)

Java provides lots of ways for you to group things together. In Chapter 12, you group things to form an array. And in Chapter 12, you group things to form a collection. In this chapter, you group things into an `enum` type. (Of course, you can't group anything unless you can pronounce `enum`. The word *enum* is pronounced "ee-NOOM," like the first two syllables of the word *enumeration.*)

Creating a complicated `enum` type isn't easy, but to create a simple `enum` type, just write a bunch of words inside a pair of curly braces. Listing 8-1 defines an `enum` type. The name of the `enum` type is `TempScale`.

LISTING 8-1: The TempScale Type (an enum Type)

```
public enum TempScale {
    CELSIUS, FAHRENHEIT, KELVIN, RANKINE
}
```

(Author's note: In Listing 8-1, I list four different temperature scales. In previous *Java For Dummies* editions, I tried to show off by listing nine temperature scales, some with fancy Unicode characters in their names. But during my research for this edition, I stumbled upon `www.curiousnotions.com/temperature-converter` a website with formulas for 73 different temperature scales. I am truly humbled.)

When you define an enum type, two important things happen:

- **You create values.**

  Just as 13 and 151 are int values, CELSIUS and FAHRENHEIT are TempScale values.

- **You can create variables to refer to those values.**

  In Listing 8-2, I declare the fields number and scale. Just as

  ```
  double number;
  ```

  declares that a number variable is of type double,

  ```
  TempScale scale;
  ```

  declares variable scale to be of type TempScale.

  "To be of type TempScale" means that you can have values CELSIUS, FAHRENHEIT, KELVIN, or RANKINE. So, in Listing 8-2, I can give the scale variable the value FAHRENHEIT (or TempScale.FAHRENHEIT, to be more precise).

TECHNICAL STUFF

An enum type is a Java class in disguise. That's why Listing 8-1 contains an entire file devoted to one thing — namely, the declaration of an enum type (the TempScale type). Like the declaration of a class, an enum type declaration belongs in a file all its own. The code in Listing 8-1 belongs in a file named TempScale.java.

Okay, so then what is a temperature?

Each temperature consists of two things: a number and a temperature scale. The code in Listing 8-2 makes this fact abundantly clear.

LISTING 8-2: The Temperature Class

```
public class Temperature {
    private double number;

    private TempScale scale;

    public Temperature() {
        number = 0.0;
        scale = TempScale.FAHRENHEIT;
    }
```

```
    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(TempScale scale) {
        number = 0.0;
        this.scale = scale;
    }

    public Temperature(double number, TempScale scale) {
        this.number = number;
        this.scale = scale;
    }

    public void setNumber(double number) {
        this.number = number;
    }

    public double getNumber() {
        return number;
    }

    public void setScale(TempScale scale) {
        this.scale = scale;
    }

    public TempScale getScale() {
        return scale;
    }

    public String toString() {
        return String.format("%5.2f\u00B0 ", number) + scale;
    }
}
```

The code in Listing 8-2 has the usual setter and getter methods (accessor methods for the `number` and `scale` fields).

CROSS REFERENCE

For some good reading on setter and getter methods (also known as accessor methods), see Chapter 7.

SEE THE FOREST FOR THE TREES

Listing 8-2 is a whopping 35 lines long, not counting the blank lines. Despite its length the code in Listing 8-2 is pretty boring. The code follows the recommended practice of making a class's fields private. (Refer to Chapter 7.) Since `number` and `scale` are private, Listing 8-2 has to include setter and getter methods. If you don't bother to make the fields private, you don't need these setters and getters, so Listing 8-2 becomes shorter, like this:

```
public class Temperature {
    double number;

    TempScale scale;

    public Temperature() {
        number = 0.0;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(double number) {
        this.number = number;
        scale = TempScale.FAHRENHEIT;
    }

    public Temperature(TempScale scale) {
        number = 0.0;
        this.scale = scale;
    }

    public Temperature(double number, TempScale scale) {
        this.number = number;
        this.scale = scale;
    }

    public String toString() {
        return String.format("%5.2f\u00B0 ", number) + scale;
    }
}
```

In fact, when I look at Listing 8-2, this is what I see:

```
public class Temperature {
    double number;
```

```
        TempScale scale;

        /*
         * Four constructors belong here.
         * Note to self: Tell readers what a constructor
         * is in the sections to follow.
         */

        public String toString() {
            return String.format("%5.2f\u00B0 ", number) + scale;
        }
    }
```

That's the right way to think about code. Focus on the broad outline. Zoom in on the details after you've taken in the big picture.

In addition to its setters and getters, Listing 8-2 has four other method-like-looking things. Each of these method-like things has the name `Temperature`, which happens to be the same as the name of the class. None of these `Temperature` method-like things has a `return` type of any kind — not even `void`, which is the cop-out `return` type.

Each of these method-like things is called a constructor. A *constructor* is like a method, except that a constructor has a special purpose: to create new objects.

REMEMBER

Whenever the computer creates a new object, the computer executes the statements inside a constructor.

What you can do with a temperature

Listing 8-3 gives form to some of the ideas that I describe in the preceding section. In Listing 8-3, you call the constructors that are declared earlier, in Listing 8-2. Figure 8-1 shows what happens when you run all this code.

```
70.00° FAHRENHEIT
32.00° FAHRENHEIT
 0.00° CELSIUS
 2.73° KELVIN
```

FIGURE 8-1: Running the code from Listing 8-3.

LISTING 8-3: **Using the Temperature Class**

```
public class UseTemperature {

    public static void main(String[] args) {

        var temp = new Temperature();
        temp.setNumber(70.0);
        temp.setScale(TempScale.FAHRENHEIT);
        System.out.println(temp);

        temp = new Temperature(32.0);
        System.out.println(temp);

        temp = new Temperature(TempScale.CELSIUS);
        System.out.println(temp);

        temp = new Temperature(2.73, TempScale.KELVIN);
        System.out.println(temp);
    }
}
```

REMEMBER

In Listing 8-3, I use the traditional style for writing a main method with the method inside an explicit class. Code of this kind doesn't implicitly import the `java.io.IO` package's `print`, `println`, and `readln` methods. (Refer to Chapter 7.) So, Listing 8-3 makes a long-winded `System.out.println` call.

MORE FOREST; FEWER TREES

The last section of Chapter 7 describes two kinds of main methods — Java's traditional main methods and the newfangled, easy-to-read main methods in simple source files. A traditional main method looks like the one in Listing 8-3:

```
public class UseTemperature {

    public static void main(String[] args) {

        // Yada, yada, ...
        System.out.println(temp);

        // Blah, blah, blah, ...
    }
}
```

When you see a traditional main method, don't get bogged down thinking about the extra text. When all is said and done, the code in Listing 8-3 is almost the same as its simpler main method cousin:

```
void main() {
    var temp = new Temperature();
    temp.setNumber(70.0);
    temp.setScale(TempScale.FAHRENHEIT);
    println(temp);

    temp = new Temperature(32.0);
    println(temp);

    temp = new Temperature(TempScale.CELSIUS);
    println(temp);

    temp = new Temperature(2.73, TempScale.KELVIN);
    println(temp);
}
```

There are a few programming pitfalls that may arise when you change from a simple source file to a traditional, long-winded main method. But life is short, so it's better not to worry about those pitfalls. Of course, if you really want to worry, you can skip to the discussion of Java `static` keyword in Chapter 11.

In Listing 8-3, each statement of the kind

```
temp = new Temperature(blah-blah);
```

calls one of the constructors from Listing 8-2. So, by the time the code in Listing 8-3 finishes running, it creates four instances of the `Temperature` class. Each instance is created by calling a different constructor from Listing 8-2.

In Listing 8-3, the last of the four constructor calls has two parameters: `2.73` and `TempScale.KELVIN`. This isn't particular to constructor calls. A method call or a constructor call can have a bunch of parameters. You separate one parameter from another with a comma. Another name for "a bunch of parameters" is a *parameter list*.

The only rule you must follow is to match the parameters in the call with the parameters in the declaration. For example, in Listing 8-3, the fourth and last constructor call

```
new Temperature(2.73, TempScale.KELVIN)
```

has two parameters: the first of type `double` and the second of type `TempScale`. Java approves of this constructor call because Listing 8-2 contains a matching declaration. That is, the header

```
public Temperature(double number, TempScale scale)
```

has two parameters: the first of type `double` and the second of type `TempScale`. If a `Temperature` constructor call in Listing 8-3 had no matching declaration in Listing 8-2, Listing 8-3 would crash and burn. (To state things more politely, Java would display errors when you tried to compile the code in Listing 8-3.)

What does a temperature look like?

The last item in Listing 8-2 is a `toString` method. When you call a `print` or `println` method, the first thing Java does is to call `toString` on your behalf. That way, whatever you're trying to display gets turned into a string of characters — a string that can show up on a computer screen. For example, when you call `println(2 + 2)`, Java adds 2 and 2 to get the `int` value 4, then it changes the `int` value `4` into the string `"4"`, and finally, Java displays the contents of that `"4"` string on the screen.

Every class has a `toString` method, whether you write code for it or not. If you remove the `toString` declaration from Listing 8-2, Java adds a `toString` method behind the scenes. I think of this added method as an implicit `toString` method. But, for reasons that become clear in Chapter 10, this added method is called the *inherited* `toString` method.

For most classes, the inherited `toString` method isn't user-friendly. To show you what I mean, I removed my `toString` declaration from Listing 8-2. Then I ran the code in Listing 8-3. This was the output:

```
Temperature@5b480cf9
Temperature@6f496d9f
Temperature@723279cf
Temperature@10f87f48
```

My `Temperature` class's inherited `toString` method isn't helpful at all. So, in Listing 8-2, I create my own `toString` method with the format string `"%5.2f\u00B0 "`. In this string, `\u00B0` is the Unicode representation of a temperature's degree symbol (°).

CROSS REFERENCE

For some wise remarks about Unicode, refer to Chapter 5.

Constructing a temperature; a slow-motion replay

When the computer executes one of the new `Temperature` statements in Listing 8-3, Java has to decide which of the constructors in Listing 8-2 to use. Java decides by looking at the parameter list — the stuff in parentheses after the words `new Temperature`. For instance, when the computer executes

```
temp = new Temperature(32.0);
```

from Listing 8-3, Java says to itself, "The number 32.0 in parentheses is a `double` value. One of the `Temperature` constructors in Listing 8-2 has just one parameter with type `double`. The constructor's header looks like this:

```
public Temperature(double number)
```

"So I guess I'll execute the statements inside that particular constructor." Java goes on to execute the following statements:

```
this.number = number;
scale = TempScale.FAHRENHEIT;
```

As a result, you get a brand-new object whose `number` field has the value `32.0` and whose `scale` field has the value `TempScale.FAHRENHEIT`.

In the two lines of code, you have two statements that set values for the fields `number` and `scale`. Take a look at the second of these statements, which is a bit easier to understand. The second statement sets the new object's `scale` field to `TempScale.FAHRENHEIT`. You see, the constructor's parameter list is `(double number)`, and that list doesn't include a `scale` value. So whoever programmed this code had to make a decision about what value to use for the `scale` field. The programmer could have chosen `FAHRENHEIT` or `CELSIUS`, but they could also have chosen `KELVIN` or `RANKINE`. (This programmer happens to live in New Jersey, in the United States, where people commonly use the old Fahrenheit temperature scale.)

Marching back to the first of the two statements, this first statement assigns a value to the new object's `number` field. The statement uses a cute trick that you can see in many constructors (and in other methods that assign values to objects' fields). To understand the trick, take a look at Listing 8-4. The listing shows you two ways that I could have written the same constructor code.

LISTING 8-4: Two Ways to Accomplish the Same Thing

```
//Use this constructor ...

public Temperature(double whatever) {
    number = whatever;
    scale = TempScale.FAHRENHEIT;
}

// ... or use this constructor ...

public Temperature(double number) {
    this.number = number;
    scale = TempScale.FAHRENHEIT;
}

// ... but don't put both constructors in your code.
```

Listing 8-4 has two constructors in it. In the first constructor, I use two different names: `number` and `whatever`. In the second constructor, I don't need two names. Rather than make up a new name for the constructor's parameter, I reuse an existing name by writing `this.number`.

Here's what's going on in Listing 8-2:

- In the statement `this.number = number`, the name *this.number* refers to the new object's `number` field — the field that's declared near the top of Listing 8-2. (See Figure 8-2.)
- In the statement `this.number = number`, *number* (on its own, without `this`) refers to the constructor's parameter. (Again, see Figure 8-2.)

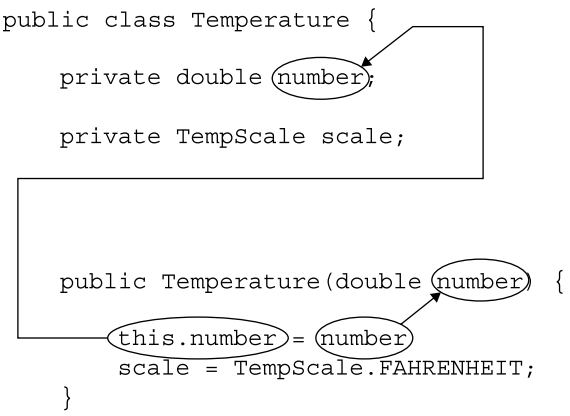


FIGURE 8-2: What `this.number` and `number` mean.

WHAT'S THIS ALL ABOUT?

Suppose that your code contains a constructor — the first of the two constructors in Listing 8-4. The `whatever` parameter is passed a number like 32.0, for instance. Then the first statement in the constructor's body assigns that value, `32.0`, to the new object's `number` field. The code works. But in writing this code, you had to make up a new name for a parameter — the name *whatever*. And the only purpose for this new name is to hand a value to the object's `number` field. What a waste! To distinguish between the parameter and the `number` field, you gave a name to something that was just momentary storage for the `number` value.

Making up names is an art, not a science. I've gone through plenty of naming phases. Years ago, whenever I needed a new name for a parameter, I picked a confusing misspelling of the original variable name. (I'd name the parameter something like `numbr` or `nuhmber`.) I've also tried changing a variable name's capitalization to come up with a parameter name. (I'd use parameter names like `Number` or `nUMBER`.) In Chapter 7, I use 1-letter variable names such as `a` for an address and `b` for an account's balance. None of these naming schemes works well — I can never remember the quirky new names I've created. The good news is that this parameter-naming effort isn't necessary. You can give the parameter the same name as the variable. To distinguish between the two, you use the Java keyword `this`.

In general, `this.`*`someName`* refers to a field belonging to the object that contains the code. In contrast, plain old *`someName`* refers to the closest place where *`someName`* happens to be declared. In the statement `this.number = number` (refer to Listing 8-2), that closest place happens to be the `Temperature` constructor's parameter list.

The default constructor

The `Temperature` class declaration in Listing 8-2 has four constructors, each with its own parameter list. What happens if you don't include all four of them? The answer isn't as straightforward as you may think. To find out what happens, try this three-step experiment:

1. **Delete all four constructors from Listing 8-2. Then try to run the code in Listing 8-3.**

   To get Listing 8-3 running, you have to delete three of the code's constructor calls. The only call that works is the parameterless `new Temperature()` call. When you don't declare any constructors, Java creates one for you. This implicit constructor has no parameters. It's called a *default constructor*.

Look back at the `Account` class in Chapter 7. In that chapter, you don't declare any constructors. And yet, you're able to call `new Account()` in each of the chapter's main methods. That's Java's default constructor at work!

CROSS REFERENCE

Java's default constructor is an implicit element. For some exciting reading about implicit elements, refer to Chapter 5.

2. **Delete three of the four constructors from Listing 8-2. The only constructor that you don't delete is the parameterless constructor.**

   When you do this, you get the same result as the one in Step 1. You don't have to let Java create an implicit default constructor for you. You can declare the parameterless constructor explicitly in your code. It's your choice.

3. **Delete three of the four constructors from Listing 8-2. The only constructor that you don't delete is the 2-parameter constructor `Temperature(double number, TempScale scale)`.**

   I call this the class's *fully parameterized constructor*. It's the constructor whose parameters include every one of the class's fields.

   When you eliminate all but the `Temperature` class's fully parameterized constructor, you can call `new Temperature(2.73, TempScale.KELVIN)`, but you can't call `new Temperature()`. Why not? What happened to Java's wonderful default constructor?

   The rule for constructors is as follows: When you declare a class and you don't explicitly declare any constructors (as in Step 1), Java creates a parameterless default constructor on your behalf. But, when you declare any constructors in your code, Java doesn't create a default constructor. So, if you want your class to have a parameterless constructor and some other constructors, you have to declare all of them (including the parameterless constructor) in your code. Pretty strange, hey!

If you find this default constructor business confusing, you can avoid thinking about it. Instead of worrying about this section's experiments, just make sure to explicitly declare all the constructors you plan to call. If you plan to call a parameterless constructor, write code for it in your class declaration. If you plan to call any parameterized constructors (fully parameterized or not), write code for these constructors in your class declaration.

Finally, if you don't enjoy writing constructor definitions as much as I do, let your IDE do the work. From your IDE's main menu, try choosing Code➪Generate➪Constructor or Source➪Generate Constructor.

REMEMBER

The terminology can be confusing. Java has *default* constructors and *inherited* methods. Both of these things appear magically when you don't explicitly define them in your code. Fortunately, you can write programs without remembering that one is called *default* and the other is called *inherited*. All you have to do is understand what's going on when you read these words.

TRY IT OUT

There's always more stuff for you to try.

SCHOOL DAYS

Create a `Student` class with a name, an ID number, a grade point average (GPA), and a major area of study. The student's name is a `String`. The student's ID number is an `int` value. The GPA is a `double` value between 0.0 and 4.0. The `Major` is an `enum` type, with values such as `COMPUTER_SCIENCE`, `MATHEMATICS`, `LITERATURE`, `PHYSICS`, and `HISTORY`.

Every student has a name and an ID number, but a brand-new student might not have a GPA or a major. Create constructors with and without GPA and `Major` parameters.

As usual, create a main method that makes use of your new `Student` class.

FLIGHT OF FANCY

Create an `AirplaneFlight` class with a flight number, a departure airport, the time of departure, an arrival airport, and a time of arrival. The flight number is an `int` value. The departure and arrival airport fields belong to an `Airport` enum type, with values corresponding to some of the official IATA airport codes. (For example, London Heathrow Airport's code is LHR; Los Angeles International Airport's code is LAX; check out `www.iata.org/publications/Pages/code-search.aspx` for a searchable database of airline codes.)

For the times of arrival and departure, use Java's `LocalTime` class. (For more on `LocalTime`, check out the `LocalTime` documents page at `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/time/LocalTime.html`.) To create a `LocalTime` object that's set to 2:15 P.M. (also known as 14:15), execute

```
LocalTime twoFifteen = LocalTime.of(14, 15);
```

To create a `LocalTime` object that's set to the current time (according to the computer's system clock), execute

```
LocalTime currentTime = LocalTime.now();
```

Every flight has a number, a departure airport, and an arrival airport. But some flights might not have departure and arrival times. Create constructors with and without departure and arrival time parameters.

Create a main method that makes use of your new `AirplaneFlight` class.

The Easy Way to Create a Class

Listing 8-2 describes the `Temperature` class in about 40 lines of code. That's a lot of code to define something that has only two properties — a number and a temperature scale. Most of the code in Listing 8-2 is routine, boilerplate stuff. When my eyes rest on the statement `this.number = number`, my brain ignores it. A devilish voice inside me says, "Don't bother me with `this.number = number`. Tell me something I don't already know." I don't expect to find anything interesting in a class declaration of this kind. So, if another programmer throws a curveball inside a class definition, I probably won't notice it. That's not good.

Maybe there's a better way to create a class. No, there isn't. I wrote this section just to complain about long class declarations.

Oops! I lied. There *is* a better way to create a class. Take a look at Listing 8-5.

LISTING 8-5: Your First Record Class

```
public record Temperature (double number, TempScale scale) {}
```

Listing 8-5 is about as small as any listing can be. And yet the code in Listing 8-5 completely defines a `Temperature` class. This `Temperature` class shares many features with the class in Listing 8-2 and even has a few features that Listing 8-2 lacks. The word `record` in Listing 8-5 makes this `Temperature` class be a special kind of class. To no one's surprise, it's called a *record class*. Every record class has certain characteristics that make life easy for Java programmers.

The code in Listing 8-6 creates instances of the record class in Listing 8-5. Figure 8-3 shows the output from Listings 8-5 and 8-6.

LISTING 8-6: Instances of a Record Class

```
void main() {

    var temp = new Temperature(70.0, TempScale.FAHRENHEIT);
    println(String.format("%5.2f\u00B0 ", temp.number()) + temp.scale());

    temp = new Temperature(2.73, TempScale.KELVIN);
    println(temp);
}
```

FIGURE 8-3: Running the code from Listing 8-6.

```
70.00° FAHRENHEIT
Temperature[number=2.73, scale=KELVIN]
```

Listing 8-6 does more or less the same things that Listing 8-3 does. The listing creates some `Temperature` instances and displays them. But record classes march to the beat of their own drummers. So Listing 8-6 isn't quite the same as the original Listing 8-3.

For one thing, a record class doesn't implicitly create a parameterless default constructor. Instead, a record class implicitly creates a fully parameterized constructor. That's why, in Listing 8-6, you see these two lines:

```
var temp = new Temperature(70.0, TempScale.FAHRENHEIT);

temp = new Temperature(2.73, TempScale.KELVIN);
```

But you don't see any of these lines:

```
// Don't add any of these lines to Listing 8-6.
var temp = new Temperature();
temp.number = 21.0;
temp.scale = TempScale.CELSIUS;

temp.setNumber(70.0);
temp.setScale(TempScale.FAHRENHEIT);
```

This section's `Temperature` class doesn't let you call `new Temperature()` and doesn't let you set the values of an existing object's `number` or `scale` fields. Once you've created an object with `new Temperature(70.0, TempScale.FAHRENHEIT)`, you can't change that object's `number` value from 70.0 to 71.5. You can't change the object's `scale` value, either.

The word *immutability* means "unchangeable," and instances of record classes are all immutable. Contrary to popular opinion, immutability is a good thing. At least, the immutability of an object is a good thing. If an object's field values never change, you know what those values are from the creation of the object until the end of the program's run. Other programmers can't add code that changes values and fouls up your logic.

But remember — this immutability business doesn't apply to variables that refer to record class instances. Look at the `temp` variable in Listing 8-6. During a run of this code, the `temp` variable refers to two different objects. One is a comfortable room temperature, and the other is the temperature of our chilly cosmic microwave background. In Listing 8-6, there's nothing to keep you from changing the object that `temp` refers to. But once you've created an object, that object's fields can't be changed.

What if you want to print a `Temperature` instance's `number` field? The expressions `temp.number` and `temp.getNumber()` won't work. But you can follow the pattern in the first `println` call of Listing 8-6. You can refer to `temp.number()` and `temp.scale()`. You type the name of the field followed by an empty pair of parentheses. That's the way you get the value of a record class's field.

WARNING

This differs from the conventions for classes that aren't record classes, so you have to be careful. (And I have to be careful. I've gotten this wrong many times.)

The last line in Figure 8-3 shows what you get when you don't explicitly format a record class instance. Java quietly concocts a decent-looking `toString` method. You don't see that `toString` method in your code. But, when you call `println(temp)` in Listing 8-6, Java displays the `temp` object's `number` and `scale` values. (Refer to Figure 8-3.) That's good!

Listing 8-5 shows you how to define a class with only one line of code. Impressive as that is, you may want to create a record class that includes some customized method declarations. Listing 8-7 adds the `toString` method to a `Temperature` record class.

LISTING 8-7: Another Record Class

```
public record Temperature (double number, TempScale scale) {

    public String toString() {
        return String.format("%5.2f\u00B0 ", number) + scale;
    }
}
```

With the `toString` method in Listing 8-7, a call to `println(temp)` displays a nicely formatted Fahrenheit room temperature. Figure 8-4 shows you what you get when you run the main method in Listing 8-6 against the `Temperature` class in Listing 8-7.

FIGURE 8-4: Two nicely formatted temperatures.

Constructors don't have to be dull

Here's a quote from somewhere near the start of this chapter: ". . . what if you're interested in doing more than filling fields? Perhaps, when the computer creates a new object, you have a whole list of jobs for the computer to carry out." Okay, what-if?

With the `Temperature` classes in previous sections, a programmer can create temperatures that are colder than absolute zero. And, in physics, that's a definite no-no. Listing 8-8 has a constructor that makes this science fiction scenario impossible.

LISTING 8-8: **It's Cold Outside!**

```
public record Temperature(double number, TempScale scale) {

    public Temperature(double number, TempScale scale) {

        this.number = Math.max(number,
                switch (scale) {
                    case CELSIUS -> -273.15;
                    case FAHRENHEIT -> -459.6;
                    case KELVIN, RANKINE -> 0.0;
                }
        );
        this.scale = scale;
    }

    public String toString() {
        return String.format("%5.2f\u00B0 ", number) + scale;
    }
}
```

When you use Listing 8-8 and run the following main method, you get the result shown in Figure 8-5.

```
void main() {
    println(new Temperature(-1000.0, TempScale.FAHRENHEIT));

    println(new Temperature(-1000.0, TempScale.CELSIUS));

    println(new Temperature(-1000.0, TempScale.KELVIN));

    println(new Temperature(2.73, TempScale.KELVIN));
}
```

```
-459.60° FAHRENHEIT
-273.15° CELSIUS
 0.00° KELVIN
 2.73° KELVIN
```

FIGURE 8-5: You're getting warmer!

In Listing 8-8, the `max` method (a member of Java's `Math` class) returns the larger of two numbers. For example, when `number` is –1000.00 and `scale` is `FAHRENHEIT`, the `switch` expression in Listing 8-8 returns –459.6. So `this.number` becomes `Math.max(-1000.00, -459.6)`, which is `-459.6`. The new object's temperature is the lowest it can possibly be, and no lower. (Just writing about it makes me want to bundle up!)

ENUM TYPES AND SWITCHES

Listings 8-2 and 8-3 contain long-winded names such as `TempScale.FAHRENHEIT` and `TempScale.CELSIUS`. Names such as `FAHRENHEIT` and `CELSIUS` belong to my `TempScale` type (the type defined in Listing 8-1). These names have no meaning outside of my `TempScale` context. (If you think I'm being egotistical with this "no meaning outside of my context" remark, try deleting the `TempScale.` part of `TempScale.FAHRENHEIT` in Listing 8-2. Suddenly, Java tells you that your code contains an error.)

Java is normally fussy about type names and dots. But when the stewards of Java created `enum` types, they decided that `enum` types in `switch` structures deserve special treatment. You can use an `enum` value to decide which case to use in a `switch` statement or `switch` expression. When you do this, you don't have to use the `enum` type name in the case expressions. For example, in Listing 8-8, the first case value is `CELSIUS`, not `TempScale.CELSIUS`.

In newer versions of Java, you can write `CELSIUS` or `TempScale.CELSIUS` inside a `switch`. But in older versions, you have to write the short name `CELSIUS`. The longer name `TempScale.CELSIUS` causes havoc in a `switch` structure's case.

Chapter 5 introduces two kinds of `switch` structures: the `switch` statement and the `switch` expression. A `switch` statement does something (such as print a message), and a `switch` expression has a value (such as `-273.15`). Java's rules for `switch` expressions differ slightly from the rules for `switch` statements.

Here's a perfectly good `switch` statement:

```
switch (scale) {
    case CELSIUS -> println("degrees Celsius");
    case FAHRENHEIT -> println("degrees Fahrenheit");
}
```

If the `temp` object's scale is `KELVIN` or `RANKINE`, this `switch` statement does nothing. No harm; no foul.

But the `switch` in Listing 8-8 is a `switch` expression. In the call to `Math.max`, the `switch` expression's value (either `-273.15`, `-459.6`, or `0.0`) gets compared with the `number` field's value. So the `switch` expression better darn well return a value. In fact, when you write a `switch` expression with an `enum` value, you must account for each of the `enum` value's possibilities. You can cop out by doing something like this:

```
println(switch (scale) {
    case CELSIUS -> "degrees Celsius";
    case FAHRENHEIT -> "degrees Fahrenheit";
    default -> "";
});
```

But the following code makes Java cry out in pain:

```
// Error! Error! Error!
println(switch (scale) {
    case CELSIUS -> "degrees Celsius";
    case FAHRENHEIT -> "degrees Fahrenheit";
});
```

The error message this code generates is:

```
'switch' expression does not cover all possible input values.
```

Equality for all!

This chapter's "What does a temperature look like?" section tells you that every class has a `toString` method. If you don't write a `toString` method, Java creates an implicit `toString` method for you. It's called an inherited `toString` method.

The same is true of Java's `equals` methods. Every class has an `equals` method. An example in Chapter 5 shows how nice Java's inherited `equals` method works for `String` values. In fact, to find out if `myName` and `yourName` contain the same characters, you shouldn't write

```
if (myName == yourName)       // No!
```

Instead, you should write

```
if (myName.equals(yourName))  // Yes
```

The `String` class's `equals` method is very useful, but the `String` class's `equals` method isn't typical. Inherited `equals` methods don't work well for arbitrary Java classes. Look at the main method in Listing 8-9. Figure 8-6 shows you the output when I run this main method along with the code in Listing 8-3.

LISTING 8-9: **Comparing Temperatures**

```
void main() {
    var freezing1 = new Temperature(0.0, TempScale.CELSIUS);
    var freezing2 = new Temperature(0.0, TempScale.CELSIUS);
    println(freezing1 + " equals " + freezing2 + " ?");
    println(freezing1.equals(freezing2));
    println("");

    var freezing3 = new Temperature(32.0, TempScale.FAHRENHEIT);
    println(freezing1 + " equals " + freezing3 + " ?");
    println(freezing1.equals(freezing3));
}
```

```
 0.00° CELSIUS equals 0.00° CELSIUS ?
false

 0.00° CELSIUS equals 32.00° FAHRENHEIT ?
false
```

FIGURE 8-6: You've got to be kidding!

In Listing 8-9, the `freezing1` and `freezing2` variables refer to two different objects. So, with the class Listing 8-3, Java's inherited `equals` method says they're not equal. That's disappointing, but what else can you expect from the `Temperature` class in Listing 8-3? It's not a record class.

Record classes behave better than other kinds of classes. When you define `Temperature` to be a record class, the inherited `toString` method gives something like `Temperature[number=2.73, scale=KELVIN]`. (See Figure 8-3.) That's good! A record class's inherited `toString` method returns a reasonable representation of an object and its fields.

So, maybe the record class in Listing 8-7 has a nice inherited `equals` method. Wha' da' ya' think? Figure 8-7 shows what you get when you run the main method in Listing 8-9 along with the record class in Listing 8-7.

```
 0.00° CELSIUS equals 0.00° CELSIUS ?
true

 0.00° CELSIUS equals 32.00° FAHRENHEIT ?
false
```

FIGURE 8-7: That's a little bit better.

If you define `Temperature` to be a record class, Java's `equals` method does an "okay" job. With two 0°-Celsius objects, the `equals` method returns `true`. That's better than what you get with a class that's not a record class. But it's not the best. The inherited `equals` method doesn't know how to convert between temperature scales.

To solve this problem, you write your own `equals` method. Listing 8-10 shows you how.

LISTING 8-10: **Comparing Temperatures**

```
public record Temperature (double number, TempScale scale) {

    public boolean equals(Temperature other) {
        var number1 = this.toCelsius();
        var number2 = other.toCelsius();
        return number1 == number2;
    }

    public double toCelsius() {
        return switch (scale) {
            case CELSIUS -> number;
```

(continued)

LISTING 8-10: *(continued)*

```
            case FAHRENHEIT -> (number - 32) * 5.0 / 9.0;
            case KELVIN -> number - 273.15;
            case RANKINE -> (number - 491.67) * 5.0 / 9.0;
        };
    }

    public String toString() {
        return String.format("%5.2f\u00B0 ", number) + scale;
    }
}
```

Combining the main method in Listing 8-9 with the `Temperature` class in Listing 8-10, you get the output shown in Figure 8-8.

```
 0.00° CELSIUS equals 0.00° CELSIUS ?
true

 0.00° CELSIUS equals 32.00° FAHRENHEIT ?
true
```

FIGURE 8-8: Perfect!

Lo and behold! The output in Figure 8-8 is correct in both cases. It helps to write your own `equals` method.

TRY IT OUT

So, you thought you were finished with this chapter about constructors! Well, there's more to do if you want to practice coding.

IT ISN'T WHAT IT ISN'T

Modify the code in Listing 8-2 so that none of the constructors lets you create a less-than-absolute-zero temperature. While you're at it, modify the setter methods, too.

KEEP IT SHORT AND SWEET

Change the code you wrote in this chapter's "School days" and "Flight of fancy" sections. Turn both classes into record classes.

IN THIS CHAPTER

- Using data on your hard drive
- Writing code to access the hard drive
- Troubleshooting input/output behavior

Chapter 9

Piles of Files: Dealing with Information Overload

Consider these scenarios:

- You're a business owner who handles hundreds of invoices each day. You store invoice data in a file on your hard drive. You need customized code to sort and classify the invoices.
- You're an astronomer with data from scans of the night sky. When you're ready to analyze a chunk of data, you load the chunk onto your computer's hard drive.
- You're the author of a popular self-help book. Last year's fad was the Self-Mirroring Method. This year's craze is the Make Your Cake System. You can't modify your manuscript without converting it to the publisher's new specifications. You need software to make the task bearable.

Each situation calls for a new computer program, and each program reads from a large data file. On top of all of that, each program creates a brand-new file containing bright, shiny results.

In previous chapters, the examples get input from the keyboard and send output to a text window. That's fine for small tasks, but you can't have the computer prompt you for each bit of night sky data. For big problems, you need lots of data, and the best place to store the data is on a computer's hard drive.

What It Means to Be an Employee

When you write an object-oriented program, you start by thinking about the data. You're writing about accounts. So, what's an account? You're writing code to handle button clicks. So, what's a button? You're writing a program to send payroll checks to employees. What's an employee?

In this chapter's first example, an employee is someone with a name and a job title. Sure, employees have other characteristics, but for now, I stick to the basics. The code in Listing 9-1 defines what it means to be an employee.

LISTING 9-1: What Is an Employee?

```
public class Employee {
    private String name;
    private String jobTitle;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }

    public String getJobTitle() {
        return jobTitle;
    }
```

```
    public void cutCheck(double amountPaid) {
        System.out.print("Pay to the order of " + name);
        System.out.print(" (");
        System.out.print(jobTitle);
        System.out.print(") ***$");
        System.out.println(String.format("%,.2f", amountPaid));
    }
}
```

WHAT HAPPENS WHEN I LISTEN TO OTHER PEOPLE

The curious reader may be interested in some history concerning Listing 9-1. In this book's first few editions, the `Employee` class listing looked like this:

```
public class Employee {
    String name;
    String jobTitle;

    public void cutCheck(double amountPaid) {
        System.out.print("Pay to the order of " + name);
        System.out.print(" (");
        System.out.print(jobTitle);
        System.out.print(") ***$");
        System.out.println(String.format("%,.2f", amountPaid));
    }
}
```

The `Employee` class code was short and sweet. But a colleague of mine read the code and remarked that "real" Java programmers wouldn't write code this way. They would make the `name` and `jobTitle` fields private. They'd compensate for the use of private fields by adding public getter and setter methods.

At first, I paid no attention to my colleague. I insisted that each of my book's examples be as short and simple as possible. I wanted readers to see the bare-bones concepts without fuss or fanfare. But, one or two editions later, I caught the "make programs look professional" disease. I made the book's examples longer and more complicated. That's why this edition's Listing 9-1 has more moving parts than the version in this sidebar.

(continued)

(continued)

I confess, dear reader! I'm ambivalent about my tendency to create more elaborate examples. But I also realize that first impressions make a difference. You're entitled to start your Java journey with code that's written the "right way." So, here and in other chapters, I let you take your pick. This sidebar's code illustrates *essentially* the same concepts as the code in Listing 9-1. "Each employee has a name, a job title, and a way of cutting a check." That's the story in a nutshell.

If looking at Listing 9-1 gives you a headache, bypass Listing 9-1 and look at this sidebar's code instead. Once you've become comfortable with the notion of an `Employee` class, you can return to Listing 9-1. Then, and only then, should you think about private fields, public accessor methods, and all the other gems that make professional programmers so very happy.

The `Employee` class in Listing 9-1 has seven members. Two of these members are fairly simple: Each employee has a `name` field and a `jobTitle` field.

The other five `Employee` members are methods. For example, each employee has four methods to handle the values of the employee's name and job title. These methods are `setName`, `getName`, `setJobTitle`, and `getJobTitle`. I explain methods like these (getter and setter methods) in Chapter 7.

On top of all of that, each employee has a `cutCheck` method. The idea is that the method that writes payroll checks has to belong to one class or another. Because most of the information in the payroll check is customized for a particular employee, you may as well put the `cutCheck` method inside the `Employee` class.

REMEMBER

The `print`, `println`, and `readln` methods that I use in many examples didn't become part of Java until September 2024, and then only as a preview feature. Until then, the easiest way to display text on the screen was to call the `System.out.println` method. (See Listing 9-1.)

Putting your class to good use

The `Employee` class in Listing 9-1 has no `main` method, so there's no starting point for executing code. To fix this deficiency, the programmer writes a separate program with a `main` method and uses that program to create `Employee` instances. Listing 9-2 shows a class with a `main` method — one that puts the code in Listing 9-1 to the test.

LISTING 9-2: **The Check Is in the Mail**

```
import java.util.Scanner;

public class PayOneEmployee {

    public static void main(String[] args) {
        var keyboard = new Scanner(System.in);

        var name = keyboard.nextLine();
        var jobTitle = keyboard.next();

        var anEmployee = new Employee();
        anEmployee.setName(name);
        anEmployee.setJobTitle(jobTitle);

        var pay = keyboard.nextDouble();

        anEmployee.cutCheck(pay);
    }
}
```

For a run of the code in Listing 9-1, see Figure 9-1.

```
Barry Burd
CEO 5000.00
Pay to the order of Barry Burd (CEO) ***$5,000.00
```

FIGURE 9-1: Barry gets paid.

SIMPLIFY, SIMPLIFY!

No matter what you do, there's always a more difficult way of doing it.

Listing 9-2 uses the traditional way of writing main methods. In the old style, every method must be inside an explicit class, every main method must be public and static, every main method must have a `String[] args` parameter, and so on, and so forth. If you use the newer features (available as preview features in Java 23), you get this simplified code:

```
void main() {
    var keyboard = new Scanner(System.in);
```

(continued)

(continued)

```
        var name = keyboard.nextLine();
        var jobTitle = keyboard.next();

        var anEmployee = new Employee();
        anEmployee.setName(name);
        anEmployee.setJobTitle(jobTitle);

        var pay = keyboard.nextDouble();

        anEmployee.cutCheck(pay);
    }
```

If you're willing to change the way a user interacts with the program, you can make it even simpler:

```
    void main() {
        var name = readln("Name? ");
        var jobTitle = readln("Job title? ");

        var anEmployee = new Employee();
        anEmployee.setName(name);
        anEmployee.setJobTitle(jobTitle);

        var pay = Double.parseDouble(readln("Pay? "));

        anEmployee.cutCheck(pay);
    }
```

Both of these main methods play nicely with Listing 9-1.

TIP

Back in Listing 9-1, notice the comma inside the `%,.2f` placeholder. The comma tells the program to use *grouping separators.* That's why, in Figure 9-1, you see `5,000.00` instead of `5000.00`. If you run the same program with the statement

```
java.util.Locale.setDefault(new java.util.Locale("kl", "GL"));
```

you get the Greenland Kalaallisut version of five thousand, which is `5.000,00`.

She reads keystrokes from the keyboard

In Listing 9-2, the code

```
import java.util.Scanner;
```

```
        var keyboard = new Scanner(System.in);

        int name = keyboard.nextLine();
```

performs almost the same function as this single statement:

```
var name = IO.readln("");  // or, in Java 24 and beyond, IO.readln();
```

The code gets whatever text the user types on a single line on the computer's keyboard. In addition, the code stores this text in the program's `name` variable. Here are some facts about this use of Java's `Scanner` class:

- **Typically (on a desktop or laptop computer), the name *System.in* stands for the keyboard.**

  To get characters from someplace other than the keyboard, you can type something other than `System.in` inside the parentheses.

  CROSS REFERENCE

  What else can you put inside the `new Scanner(. . .)` parentheses? For some ideas, see this chapter's later section "Working with Disk Files."

  In Listing 9-2, I make the arbitrary decision to give one of my variables the name `keyboard`. The name `keyboard` reminds you, the reader, that this variable refers to a bunch of plastic buttons in front of your computer. Naming something `keyboard` doesn't tell Java anything about plastic buttons or about user input. On the other hand, the name `System.in` always tells Java about those plastic buttons. The code `var keyboard = new Scanner (System.in)` in Listing 9-2 connects the name `keyboard` with the plastic buttons that we all know and love.

- **Java's `Scanner` class has several different `next`*`SomethingOrOther`* methods.**

  Here are some examples:

  - The `nextDouble` method reads a `double` value (a number containing a decimal point). Listing 9-2 calls `nextDouble` to get a payment amount.

    WARNING

    The representation of decimal points varies from one country to another. For details (and some interesting trivia), see this chapter's later sidebar "Where on Earth do you live?"

  - The `nextInt` method reads an `int` value (a whole number of some kind).
  - The `next` method reads a single word — a bunch of text up to, but not including, some white space. In Figure 9-1, the call `keyboard.next()` gets the word `CEO` and nothing else.

By the way, the term *white space* means a blank space, a tab character, the end of a line, and several other oddities.

- The `nextLine` method gets everything up to the end of a line. In Figure 9-1, a call to `nextLine` scoops up everything on one line of input (both words `Barry` and `Burd`).

  Some surprising things happen when you combine `nextLine` with other `Scanner` methods. For details, see this chapter's later section "Reading a line at a time."

CROSS REFERENCE

» **Whenever you use Java's `Scanner`, you should call the `close` method after your last `next`*`Whatever`* call.**

In Listing 9-2, the `main` method's last statement is

```
keyboard.close();
```

This statement does some housekeeping to disconnect the Java program from the computer keyboard. (The amount of required housekeeping is more than you might think!) If I omit this statement from Listing 9-2, nothing terrible happens. Java usually cleans up after itself very nicely. But using `close()` to explicitly detach from the keyboard is good practice, and some IDEs display warnings if you omit the `keyboard.close()` statement. In this book's example, I always remember to close my `Scanner` variables.

In Chapter 14, I show you a more reliable way to incorporate the `keyboard.close()` statement in your Java program.

The listings in previous chapters use Java's `readln` method to get input from the keyboard. So, why not use `readln` here in Listing 9-2? As of September 2024, with the release of Java 23, the newly-created `readln` method works only with keyboard input. If you want to get data from a disk file, you can't use the `readln` method. And no one wants to go through life without reading from a disk file. Relying exclusively on keyboard input is too confining. If you have several employees, you store employee information on a disk.

Java has many classes and methods for reading input of various kinds. The `Scanner` class from Listing 9-2 dates back to the release of Java 5.0 in 2004. A more versatile input package, `java.nio`, was born in 2002 as part of Java 1.4. And the venerable `InputStreamReader`/`BufferedReader` classes came with Java 1.0 in 1996.

In any programming language, with input of any kind, there's a lot going on under the hood. Your program sets up a connection between the code and the storage device, prepares for possible trouble reading from the device, does the reading, converts the characters it reads into the type of value that you want and,

finally, breaks your connection with the storage device. And the underlying operating system does even more work. It's a big mess.

Java's `Scanner` class doesn't offer all the options you get with the old Java 1.0 classes or with `java.nio`. In exchange for its lack of versatility, the `Scanner` class makes disk input fairly simple. Even so, disk input with `Scanner` can confuse novice programmers. So, the next section translates Listing 9-2 into a program that reads from a disk file using the `Scanner` class.

TRY IT OUT

There are always interesting things for you to try:

GO, SCANNER, GO

Write a program that reads two `double` values separated by an arithmetic operator sign and uses a `switch` structure to do the arithmetic. For example, if the user types `2.0 * 5.1`, the program outputs `10.2`.

LOCALE, LOCALE, LOCALE

A `PlaceToLive` has an address, a number of bedrooms, and an area (in square feet or square meters). Write the `PlaceToLive` class's code. Write code for a main method that creates a few `PlaceToLive` instances by assigning values to their `address`, `numberOfBedrooms`, and `area` fields. The main method also reads (from the keyboard) the cost of living in each place. For each place, your code displays the cost per square foot (or square meter) and the cost per bedroom.

Working with Disk Files

Your company is growing. In Listing 9-3, you had only one employee. In the next listing, you have three!

LISTING 9-3: **Writing Payroll Checks**

```
import java.io.File;           // These imports are optional for
import java.io.IOException;    //   a simple source file
import java.util.Scanner;      //   (like this file, for example).

void main() throws IOException {
    var diskScanner = new Scanner(new File("EmployeeInfo.txt"));

    for (int empNum = 1; empNum <= 3; empNum++) {
        var anEmployee = new Employee();
```

(continued)

LISTING 9-3: *(continued)*

```
            anEmployee.setName(diskScanner.nextLine());
            anEmployee.setJobTitle(diskScanner.nextLine());
            anEmployee.cutCheck(diskScanner.nextDouble());
            diskScanner.nextLine();
        }

        diskScanner.close();
}
```

The `for` loop in Listing 9-3 performs three iterations. Here's how the variable name *anEmployee* is reused and recycled during those iterations:

- During the first iteration, the statement `anEmployee = new Employee()` makes `anEmployee` refer to a new object.
- During the second iteration, the computer executes the same statement again. This second execution creates a new incarnation of the `anEmployee` variable that refers to a brand-new object.
- During the third iteration, all the same stuff happens again. A new `anEmployee` variable ends up referring to a third object.

The whole story is pictured in Figure 9-2.

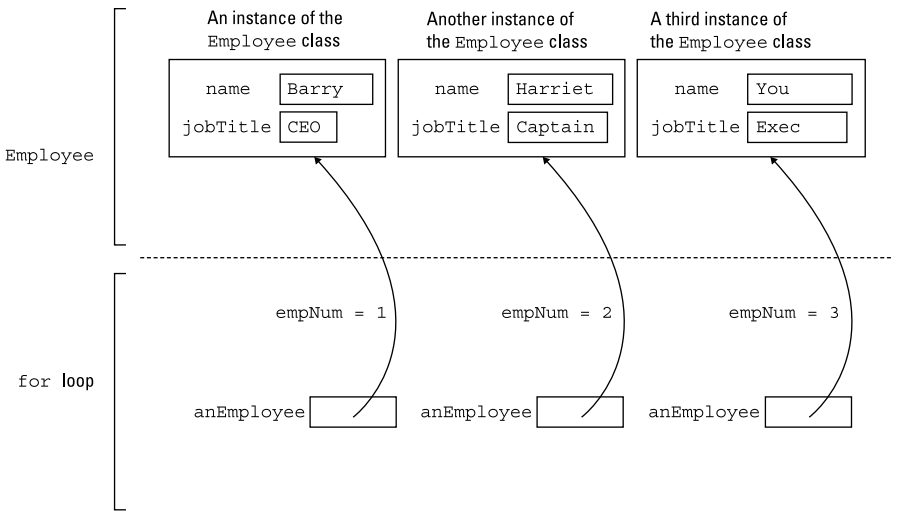


FIGURE 9-2: Three employees.

To run the code in Listing 9-3, your hard drive must contain a file named `EmployeeInfo.txt`. Fortunately, the stuff that you download from this book's website (`http://javafordummies.allmycode.com`) comes with an `EmployeeInfo.txt` file. Just copy the `EmployeeInfo.txt` file to a place where Listing 9-3 can find it.

CROSS REFERENCE

For help deciding where Listing 9-3 can find the file, see this chapter's "Where's your file?" section.

Figure 9-3 shows the `EmployeeInfo.txt` file you get when you visit this book's website and download the book's code.

```
Barry Burd
CEO
5000.00
Harriet Ritter
Captain
7000.00
Your Name Here
Honorary Exec of the Day
10000.00
```

FIGURE 9-3: An `EmployeeInfo.txt` file.

You can create your own `EmployeeInfo.txt` file with Windows Notepad or TextEdit on a Mac. (If you use TextEdit, be sure to choose Settings➪ Format➪ Plain Text.) When you type the characters in Figure 9-3, you must finish up by typing `10000.00` and then pressing Enter. (Look again at Figure 9-3 and notice how the cursor is at the start of a brand-new line.) If you forget to finish by pressing Enter, the code in Listing 9-3 will crash when you try to run it.

And remember: Grouping separators vary from one country to another. The file shown in Figure 9-3 works on a computer configured in the United States where *5000.00* means "five thousand." But the file doesn't work on a computer that's configured in what I call a "comma country" — a country where *5000,00* means "five thousand." If you live in a comma country, be sure to read this chapter's nearby sidebar "Where on Earth do you live?"

Figure 9-4 shows the output when you run Listing 9-3 using `EmployeeInfo.txt` file in Figure 9-3.

```
Pay to the order of Barry Burd (CEO) ***$5,000.00
Pay to the order of Harriet Ritter (Captain) ***$7,000.00
Pay to the order of Your Name Here (Honorary Exec of the Day) ***$10,000.00
```

FIGURE 9-4: Everybody gets paid.

WHERE ON EARTH DO YOU LIVE?

Grouping separators vary from one country to another. In the United States, *10.5* (with a period) represents ten-and-a-half, but in France, *10,5* (with a comma) represents ten-and-a-half. In India, you group digits in sets of two and three. For example, you write 1,23,45,67,890.55. Your computer's operating system stores information about the country you live in, and Java reads that information to decide how numbers should look.

This makes a big difference when you try to read `double` values using Java's `Scanner` class. To see what I mean, have a serious look at this slightly edited JShell session:

```
jshell> import java.util.Scanner
jshell> import java.util.Scanner; import java.util.Locale

jshell> var keyboard = new Scanner(System.in)
keyboard ==> java.util.Scanner

jshell> keyboard.nextDouble()
1000
$4 ==> 1000.0

jshell> Locale.setDefault(Locale.FRANCE); keyboard = new Scanner(System.in)
keyboard ==> java.util.Scanner

jshell> keyboard.nextDouble()
1000,00
$7 ==> 1000.0

jshell> keyboard.nextDouble()
1000.00
|  Exception java.util.InputMismatchException
|        at Scanner.throwFor (Scanner.java:964)
|        at Scanner.next (Scanner.java:1619)
|        at Scanner.nextDouble (Scanner.java:2590)
```

I conducted this session on a computer in the United States. The country of origin is relevant because, in response to the first `keyboard.nextDouble()` call, I type `1000.00` (with a period before the last two zeros) and Java accepts this as meaning "one thousand."

But then, in the JShell session, I call `Locale.setDefault(Locale.FRANCE)`, which tells Java to behave as if my computer is in France. When I create another `Scanner` instance and call `keyboard.nextDouble()` again, Java accepts `1000,00` (with a comma before the last two zeros) as an expression meaning *mille* (French for "one thousand"). What's

more, Java no longer accepts the period in `1000.00`. When I type `1000.00` (with a period), I get an `InputMismatchException`.

By default, Java wants you to input `double` numbers the way you normally type them in your country. If you type numbers according to another country's convention, you get an `InputMismatchException`. So, when you run the code in Listing 9-3, the numbers in your `EmployeeInfo.txt` file must use your country's format.

This brings me to the running of the code in Listing 9-3. The `EmployeeInfo.txt` file that you download from this book's website starts with the following three lines:

```
Barry Burd
CEO
5000.00
```

That last number `5000.00` has a period in it, so if your country prefers a comma in place of my United States period, you get an `InputMismatchException`. In response to this, you have two choices:

- In the downloaded `EmployeeInfo.txt` file, change the periods to commas.
- In the code of Listing 9-3, add the statement `Locale.setDefault(Locale.US)` before the `diskScanner` declaration.

And finally, if you want your output to look like your own country's numbers, you can do it with Java's `String` class. Add something like this to your code:

```
println(String.format(java.util.Locale.FRANCE, "%,.2f", 1000.00));
```

For all the details, see the API (Application Programming Interface) documentation for Java's `Formatter` class (`https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Formatter.html`) and `Locale` class (`https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Locale.html`).

Cutting a check

In previous chapters, programs read characters from the computer's keyboard. But the code in Listing 9-3 reads characters from a specific file. The file (named `EmployeeInfo.txt`) lives on your computer's hard drive.

This `EmployeeInfo.txt` file is like a word processing document. The file can contain letters, digits, and other characters. But unlike a word processing document,

the `EmployeeInfo.txt` file contains no formatting — no italics, no bold, no font sizes, nothing of that kind.

The `EmployeeInfo.txt` file contains only ordinary characters — the kinds of keystrokes that you type while you play a guessing game from Chapters 5 and 6. Of course, getting guesses from a user's keyboard and reading employee data from a disk file aren't exactly the same. In a guessing game, the program displays prompts, such as `Pick a number from 1 to 10`. The game program conducts a back-and-forth dialogue with the person sitting at the keyboard. In contrast, Listing 9-3 has no dialogue. This `DoPayroll` program reads characters from a hard drive and doesn't prompt or interact with anyone.

Repeat after me

In almost any computer programming language, reading data from a file can be tricky. You add extra lines of code to tell the computer what to do. Sometimes you can copy and paste these lines from other people's code. For example, you can follow this pattern (extracted from Listing 9-3):

```
/*
 * The pattern in Listing 9-3
 */
import java.io.File;
import java.io.IOException;
import java.util.Scanner;

void main() throws IOException {

    var scannerName = new Scanner(new File("SomeFileName"));

    //Some code goes here

    scannerName.nextInt();
    scannerName.nextDouble();
    scannerName.next();
    scannerName.nextLine();
    //Some code goes here

    scannerName.close();
}
```

You want to read data from a file. You start by imagining that you're reading from the keyboard. Put the usual `Scanner` and `next` codes into your program. Then add some extra items from the Listing 9-3 pattern:

- Add three new `import` declarations: one for `java.io.File`, another for `java.io.IOException`, and a third for `java.util.Scanner`. (With a file that has an implicitly declared class, like the file in Listing 9-3, these imports are optional.)
- Type **throws IOException** in your method's header.
- Type **new File(" ")** in your call to `new Scanner`.
- Take a file that's already on your hard drive. Type that filename inside the quotation marks.
- Take the word that you use for the name of your scanner. Reuse that word in calls to `next`, `nextInt`, `nextDouble`, and so on.
- Take the word that you use for the name of your scanner. Reuse that word in a call to `close`.

Occasionally, copying and pasting code can get you into trouble. Maybe you're writing a program that doesn't fit the simple Listing 9-3 pattern. You need to tweak the pattern a bit. But to tweak the pattern, you need to understand some of the ideas behind the pattern.

That's how the next section comes to your rescue. It covers some of these ideas.

Reading from a file

To read characters from a file, you can use the `Scanner` class and methods like `nextLine` and `nextDouble`. But in addition to these goodies, you have a few extra hurdles to jump. Here's a list:

- **You need a new `File` object.** To be more precise, you need a new instance of the API's `File` class. You get this new instance with code like

  ```
  new File("EmployeeInfo.txt")
  ```

  The stuff in quotation marks is the name of a file — a file on your computer's hard drive. The file contains characters like those shown previously in Figure 9-3.

At this point, the terminology makes mountains out of molehills. Sure, I use the phrases *new File object* and *new File instance,* but all you're doing is making `new File("EmployeeInfo.txt")` stand for a file on your hard drive. After you shove `new File("EmployeeInfo.txt")` into `new Scanner`,

```
var diskScanner = new Scanner(new File("EmployeeInfo.txt"));
```

you can forget all about the `new File` business. From that point on in the code, `diskScanner` stands for the `EmployeeInfo.txt` file on your computer's hard drive. (The name `diskScanner` stands for a file on your hard drive just as, in previous examples, the name `keyboard` stands for those buttons you press day in and day out.)

REMEMBER

Creating a `new File` object in Listing 9-3 is like creating a `new Employee` object later in the same listing. It's also like creating a `new Account` object in the examples of Chapter 7. The only difference is that the `Employee` and `Account` classes are defined in this book's examples. The `File` class is defined in Java's API.

WARNING

When you connect to a disk file with `new Scanner`, don't forget the `new File` part. If you write `new Scanner("EmployeeInfo.txt")` without `new File`, Java's compiler doesn't mind. (You don't get any warnings or error messages before you run the code.) But when you run the code, you don't get anything like the results that you expect to get.

- **You need a `throws IOException` clause.** Lots of things can go wrong when your program connects to `EmployeeInfo.txt`. For one thing, your hard drive may not have a file named `EmployeeInfo.txt`. For another, the file `EmployeeInfo.txt` may be in the wrong directory. To brace for this kind of calamity, the Java programming language takes certain precautions. The language insists that when a disk file is involved, you acknowledge the possible dangers of calling `new Scanner`.

  You can acknowledge the hazards in several possible ways, but the simplest way is to use a `throws` clause. In Listing 9-3, the `main` method's header ends with the words *throws IOException.* By adding these two words, you appease the Java compiler. It's as if you're saying "I know that calling `new Scanner` can lead to problems. You don't have to remind me." And, sure enough, adding `throws IOException` to your `main` method keeps Java from complaining. (Without this `throws` clause, you get an `unhandled exception` error message.)

CROSS REFERENCE

For the full story on Java exceptions, read Chapter 14. In the meantime, add `throws IOException` to the header of any method that calls `new Scanner(new File( . . . ))`.

Where's your file?

If Java can't find an `EmployeeInfo.txt` file, the whole project doesn't run properly. Instead, you get a `FileNotFoundException`.

In general, when you get a `FileNotFoundException`, some file that your program needs isn't available to it. This is an easy mistake to make. It can be frustrating because, to you, a file such as `EmployeeInfo.txt` may look like it's available to your program. But remember: Computers are stupid. If you make a tiny mistake, the computer can't read between the lines for you. If the filename is spelled incorrectly, the computer chokes when it tries to run your code.

Sometimes you know darn well that an `EmployeeInfo.txt` (or `whatever.xyz`) file exists on your hard drive. But when you run your program, you still get a mean-looking `FileNotFoundException`. When this happens, the file may be in the wrong directory on your hard drive. (Of course, it depends on your point of view. Maybe the file is in the right directory but your Java program is looking for the file in the wrong directory.) To diagnose this problem, add the following code to Listing 9-3:

```
var employeeInfo = new File("EmployeeInfo.txt");
System.out.println("Looking for " + employeeInfo.getCanonicalPath());
```

When you run the code, Java tells you where, on your hard drive, the `EmployeeInfo.txt` file should be.

You can decide where to put your file

With code like `new File("EmployeeInfo.txt")`, your IDE decides where to look for a file named `EmployeeInfo.txt`. But you can override this behavior by naming a file's exact location in your Java code. Code like `new File("C:\\Users\\bburd\\Documents\\EmployeeInfo.txt")` on Windows or `File("/Users/bburd/Documents/EmployeeInfo.txt")` on a Mac looks really ugly, but it works.

In the preceding paragraph, did you notice the double backslashes in `"C:\\Users\\bburd\\Documents . . ."`? If you're a Windows user, you'd be tempted to write `C:\Users\bburd\Documents . . .` with single backslashes. But in Java, the single backslash has its own, special meaning. (For example, `print("\n\nBoo!")` means, "Skip two lines and then print `Boo!`.") So, in Java, to indicate a backslash inside a quoted string, you use a double backslash instead.

Reading a line at a time

Listings 9-2 and 9-3 illustrate some useful tricks for reading data. In particular, every scanner that you create has a `nextLine` method. (You might not use this `nextLine` method, but the method is available nonetheless.) When you call a scanner's `nextLine` method, the method grabs everything up to the end of the current line of text.

In the previous paragraph, notice my careful choice of words: `nextLine` reads everything "up to the end of the current line." Unfortunately, what it means to read up to the end of the current line isn't always what you think it means. Intermingling `nextInt`, `nextDouble`, and `nextLine` calls can be messy. You have to watch what you're doing and check your program's output carefully.

To understand all of this, you need to be painfully aware of a data file's line breaks. Think of a line break as an extra character, stuck between one line of text and the next. Then imagine that calling `nextLine` means to read everything up to and including the next line break.

Figure 9-5 contains an annotated version of the first several lines in Figure 9-3. Notice what you do to read `5000.00` followed by `Harriet Ritter`:

- You call `nextDouble` to get `5000.00` and then,
- You call `nextLine`, thinking that you'll get `Harriet Ritter` from the call. But, instead, you get the line break after `5000.00`.

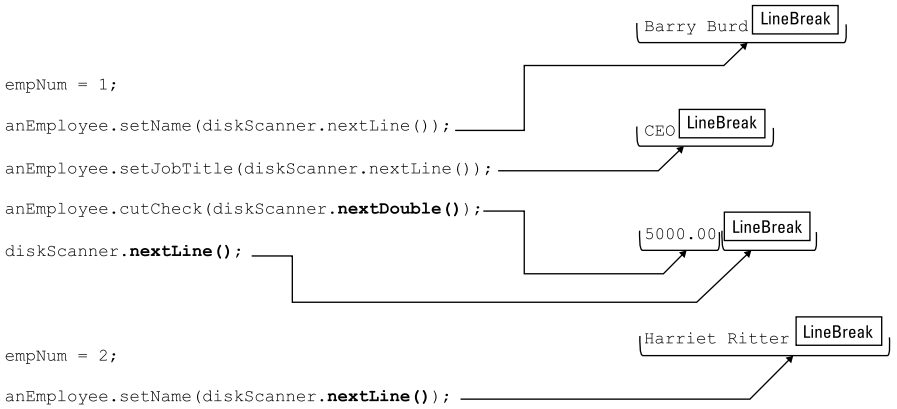


FIGURE 9-5: Calling `nextDouble` and `nextLine`.

So, after reading the number `5000.00`, you need *two* calls to `nextLine` in order to scoop up the name `Harriet Ritter`. The mistake that I usually make is to forget the first of those two calls.

WARNING

Look again at the file in Figure 9-3. For this section's code to work correctly, you must have a line break after the last `10000.00`. If you don't, a final call to `nextLine` makes your program crash and burn. The error message reads `NoSuchElementException: No line found`.

TECHNICAL STUFF

I'm always surprised by the number of quirks that I find in each programming language's scanning methods. For example, the first `nextLine` that reads from the file in Figure 9-3 devours `Barry Burd[LineBreak]` from the file. But that `nextLine` call delivers `Barry Burd` (with no line break) to the running code. So `nextLine` looks for a line break, and then `nextLine` loses the line break. Yes, this is a subtle point. And no, this subtle point hardly ever causes problems for anyone.

TECHNICAL STUFF

If this business about `nextDouble` and `nextLine` confuses you, please don't put the blame on Java. Mixing input calls is delicate work in any computer programming language. And the truly nasty thing is that each programming language approaches the problem a little differently. What you find out about `nextLine` in Java helps you understand the issues when you get to know C++ or Visual Basic, but it doesn't tell you all the details. Each language's details are unique to that language. (Yes, it's a big pain. But because all computer programmers become rich and famous, the pain eventually pays off.)

Clean up after yourself

To the average computer user, a keyboard doesn't feel anything like a file stored on a computer's hard drive. But disk files and keyboard input have a lot in common. In fact, a basic principle of computer operating systems dictates that any differences between two kinds of input be, for the programmer, as blurry as possible. As a Java programmer, you should treat disk files and keyboard input almost the same way. That's why Listing 9-3 contains a `diskScanner.close()` call.

When you run a Java program, you normally execute the `main` method's statements, starting with the first statement in the method body and ending with the last statement in the method body. You take detours along the way, skipping past `else` parts and diving into method bodies, but basically you finish executing statements at the end of the `main` method. That's why, in Listing 9-3, the call to `close` is at the end of the `main` method's body. When you run the code in Listing 9-3, the last thing you do is disconnect from the disk file. And, fortunately, that disconnection takes place after you've executed all the `nextLine` and `nextDouble` calls.

ON THE RECORD

TRY IT OUT

Previously in this chapter, you create instances of your own `PlaceToLive` class and display information about those instances. Modify the text-based version of your code so that it gets each instance's characteristics (address, number of bedrooms, and area) from a disk file.

IN THIS CHAPTER

- Adding new life to old code
- Tweaking your code
- Making changes without spending a fortune

Chapter 10

Saving Time and Money: Reusing Existing Code

reuse /ree-YOOSS/ *noun* The act of using something for the *n*th time, where *n* is greater than 1. Example: "Reuse of the material in *Java For Dummies*, 9th Edition, is strictly prohibited."

Reuse is good but, in many situations, reuse is a rarity. For example, in the United States, the Department of Agriculture estimates that 30 to 40 percent of the nation's food goes to waste. And, according to one source, only 9 percent of the world's plastics are recycled.

Information goes to waste too. When I revised this chapter for the eighth edition, I decided not to reuse the seventh edition's chapter introduction. (That introduction's made-up story isn't amusing in the least.) In desperation, I looked at some of my other books to find an introduction that I could reuse for this chapter. No luck! I came up empty-handed.

I can't even reuse paragraphs to explain common concepts. My description of Java classes from another book wouldn't work well in this book. I've even experienced times when I had to scrap sections in several chapters because of a small update in Chapter 3.

No doubt about it! Reuse is a precious commodity, and it's in very short supply. Wouldn't it be nice if every piece of software did just what you wanted it to do? In an ideal world, you could buy a program, make it work right away, plug it seamlessly into new situations, and update it easily whenever your needs change. Unfortunately, software of this kind doesn't exist. (Nothing of this kind exists.) The truth is that no matter what you want to do, you can find software that does some of it, but not all of it.

This is one of the reasons why object-oriented programming has been successful. For years, companies were buying prewritten code, only to discover that the code didn't do what they wanted it to do. So, what did the companies do about it? They started messing with the code. Their programmers dug deep into the program files, changed variable names, moved subprograms around, reworked formulas, and generally made the code worse. The reality was that if a program didn't already do what you wanted it to do (even if it did something ever so close to what you wanted), you could never improve the situation by mucking around inside the code. The best option was always to chuck the whole program (expensive as that was) and start all over again. What a sad state of affairs!

With object-oriented programming, a big change has come about. At its heart, an object-oriented program is made to be modified. With correctly written software, you can take advantage of features that are already built-in, add new features of your own, and override features that don't suit your needs. And the best part is that the changes you make are clean — no clawing and digging into other people's brittle program code. Instead, you make nice, orderly additions and modifications without touching the existing code's internal logic. It's the ideal solution.

Defining Subclasses (What It Means to Be a Full-Time or Part-Time Employee)

This time last year, your company paid $10 million for a piece of software. (Too bad! You didn't know you could find that software in Chapter 9 of *Java For Dummies,* 9th Edition.)

The software came in the form of an `Employee.class` file. People at Burd Brain Consulting (the company that created the software) didn't want you to know about the innards of the software. (They didn't want you to steal their ideas.) So, you don't have the Java program file that the software came from. In other words, you *don't* have the `Employee.java` file in Listing 9-1 (over in Chapter 9) and Listing 10-1.

LISTING 10-1: What Is an Employee?

```
public class Employee {
    private String name;
    private String jobTitle;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }

    public String getJobTitle() {
        return jobTitle;
    }

    public void cutCheck(double amountPaid) {
        System.out.print("Pay to the order of " + name);
        System.out.print(" (");
        System.out.print(jobTitle);
        System.out.print(") ***$");
        System.out.println(String.format("%,.2f", amountPaid));
    }
}
```

You can run the bytecode in the `Employee.class` file. You can also read the documentation in a web page named *Employee.html.* But you can't see the statements inside the `Employee.java` program, and you can't change any of the program's code.

REMEMBER

Chapter 2 describes the difference between filenames ending with `.java` and those ending with `.class`. To write code that uses the `Employee` class, you don't need the `Employee.java` file on your hard drive. All you need is the file `Employee.class`.

Since this time last year, your company has grown. Unlike in the old days, your company now has two kinds of employees: full-time and part-time. Each full-time employee is on a fixed, weekly salary. (If the employee works nights and weekends, then in return for this monumental effort, the employee receives a hearty handshake.) In contrast, each part-time employee works for an hourly

wage. Your company deducts an amount from each full-time employee's paycheck to pay for the company's benefits package. Part-time employees, however, don't get benefits.

The question is whether the software that your company bought last year can keep up with the company's growth. You invested in a helpful program to handle employees and their payroll, but the program doesn't differentiate between your full-time and part-time employees. You have several options:

- **Call your next-door neighbor, whose 12-year-old child knows more about computer programming than anyone in your company.** Get this uppity little brat to take the employee software apart, rewrite it, and hand it back to you with all the changes and additions your company requires.

  On second thought, you can't do that. No matter how smart that kid is, the complexities of the employee software will probably confuse the kid. By the time you get the software back, it'll be filled with bugs and inconsistencies. Besides, you don't even have the `Employee.java` file to hand to the kid. All you have is the `Employee.class` file, which is bytecode. And bytecode can't be read or modified with a text editor. (See Chapter 2.) Besides, your kid just beat up the neighbor's kid. You don't want to give your neighbor the satisfaction of seeing you beg for the whiz kid's help.

- **Scrap the $10 million employee software.** Persuade someone in your company to rewrite the software from scratch.

  In other words, say goodbye to your time and money.

- **Write a new front end for the employee software.** That is, build a piece of code that does some preliminary processing on full-time employees and then hands the preliminary results to your $10 million software. Do the same for part-time employees.

  This idea could be decent or spell disaster. Are you sure that the existing employee software has convenient *hooks* in it? (That is, does the employee software contain entry points that allow your front-end software to easily send preliminary data to the expensive employee software?) *Remember:* This plan treats the existing software as one big, monolithic lump, which can become cumbersome. Dividing the labor between your front-end code and the existing employee program is difficult. And, if you add layer upon layer to existing black box code, you'll probably end up with a fairly inefficient system.

- **Call Burd Brain Consulting, the company that sold you the employee software.** Tell Dr. Burd that you want the next version of his software to differentiate between full-time and part-time employees.

"No problem," says Dr. Burd. "It'll be ready by the start of the next fiscal quarter." That evening, Dr. Burd makes a discreet phone call to his next-door neighbor. . . .

» **Create two new Java classes named *FullTimeEmployee* and *PartTimeEmployee.*** Have each new class extend the existing functionality of the expensive `Employee` class, but have each new class define its own, specialized functionality for certain kinds of employees.

Way to go! Figure 10-1 shows the structure that you want to create.

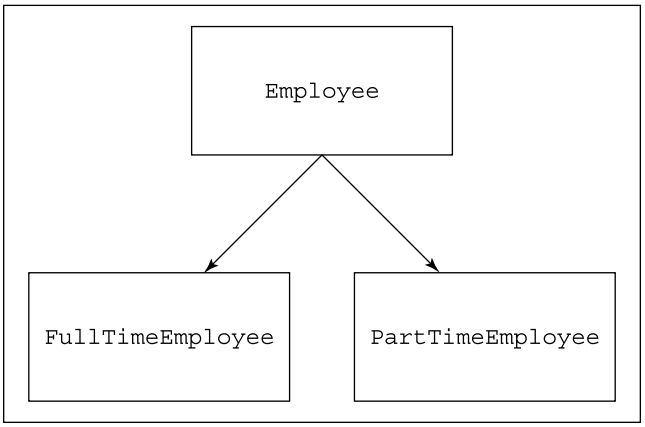


FIGURE 10-1: The `Employee` class family tree.

Creating a subclass

Listing 10-1 defines an `Employee` class. You may not have access to the `Employee.java` file, but if you have the compiled `Employee.class` file and some documentation about the code, you can extend the definition of an `Employee` to create new, more specialized classes. So, in Listing 10-2, I define a new class: a `FullTimeEmployee` class.

LISTING 10-2: **What Is a FullTimeEmployee?**

```
public class FullTimeEmployee extends Employee {
    private double weeklySalary;
    private double benefitDeduction;

    public void setWeeklySalary(double weeklySalary) {
        this.weeklySalary = weeklySalary;
    }
```

(continued)

LISTING 10-2: ***(continued)***

```
    public double getWeeklySalary() {
        return weeklySalary;
    }

    public void setBenefitDeduction(double benefitDeduction) {
        this.benefitDeduction = benefitDeduction;
    }

    public double getBenefitDeduction() {
        return benefitDeduction;
    }

    public double findPaymentAmount() {
        return weeklySalary - benefitDeduction;
    }
}
```

Looking at Listing 10-2, you can see that each instance of the `FullTimeEmployee` class has two fields: `weeklySalary` and `benefitDeduction`. But are those the only fields that each `FullTimeEmployee` instance has? No, they're not. The first line of Listing 10-2 says that the `FullTimeEmployee` class extends the existing `Employee` class. This means that in addition to having a `weeklySalary` and a `benefitDeduction`, each `FullTimeEmployee` instance has two other fields: `name` and `jobTitle`. These two fields come from the definition of the `Employee` class, which you can find in Listing 10-1.

In Listing 10-2, the magic word is *extends.* When one class extends an existing class, the extending class implicitly inherits functionality that's defined in the existing class. So, the `FullTimeEmployee` class *inherits* the `name` and `jobTitle` fields. The `FullTimeEmployee` class also inherits all the methods that are declared in the `Employee` class: `setName`, `getName`, `setJobTitle`, `getJobTitle`, and `cutCheck`.

The `FullTimeEmployee` class is a *subclass* of the `Employee` class. That means the `Employee` class is the *superclass* of the `FullTimeEmployee` class. You can also talk in terms of blood relatives: The `FullTimeEmployee` class is the *child* of the `Employee` class, and the `Employee` class is the *parent* of the `FullTimeEmployee` class.

IGNORANCE IS BLISS

I'm a big believer in the value of ignoring details. When I look at Listing 10-2, I ignore most of the text. I see only a few words surrounded by some curly braces.

```
public class FullTimeEmployee extends Employee {
    double weeklySalary;
    double benefitDeduction;

    double findPaymentAmount() {
        return weeklySalary - benefitDeduction;
    }
}
```

I see a class with two fields and a `findPaymentAmount` method. I also see the words `extends Employee`, and I say to myself "The words `extends Employee` don't appear in any of the previous chapters. So, the rest of this chapter should be about words like `extends Employee`." And, indeed, the rest of this chapter is about extending classes.

This sidebar's code is much shorter than the code in Listing 10-2. But that's because this sidebar's code doesn't do things exactly the same way as the code in Listing 10-2. In particular, the code in Listing 10-2 has private fields with accessor methods. So, later in this chapter (Listing 10-5, in fact), you see the following lines:

```
var ftEmployee = new FullTimeEmployee();
ftEmployee.setWeeklySalary(5000.00);
```

The second of these lines calls the `setWeeklySalary` method in Listing 10-2. That's all well and good, but if you use this sidebar's simpler `FullTimeEmployee` class, you write the following code instead:

```
var ftEmployee = new FullTimeEmployee();
ftEmployee.weeklySalary = 5000.00;
```

Instead of `setWeeklySalary(5000.00)`, you write `weeklySalary = 5000.00`. The first time you see these two code snippets, you may have trouble distinguishing one from the other. But when you've had practice writing several Java programs, you stop fretting about parentheses and equal signs. You simply remember that a `FullTimeEmployee` has a weekly salary, a benefit deduction, and can find its own payment amount. Also, in some way, shape, or form, a full-time employee `extends Employee`.

Several names from Listing 10-1 are implicitly defined in Listing 10-2. It's almost (but not quite) as if the `FullTimeEmployee` class were defined by the code in Listing 10-3.

LISTING 10-3: Fake (but Informative) Code

```
public class FullTimeEmployee {
    private String name;
    private String jobTitle;
    private double weeklySalary;
    private double benefitDeduction;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setJobTitle(String jobTitle) {
        this.jobTitle = jobTitle;
    }

    public String getJobTitle() {
        return jobTitle;
    }

    public void setWeeklySalary(double weeklySalary) {
        this.weeklySalary = weeklySalary;
    }

    public double getWeeklySalary() {
        return weeklySalary;
    }

    public void setBenefitDeduction(double benefitDeduction) {
        this.benefitDeduction = benefitDeduction;
    }

    public double getBenefitDeduction() {
        return benefitDeduction;
    }

    public double findPaymentAmount() {
        return weeklySalary - benefitDeduction;
    }
```

```
    public void cutCheck(double amountPaid) {
        System.out.print("Pay to the order of " + name);
        System.out.print(" (");
        System.out.print(jobTitle);
        System.out.print(") ***$");
        System.out.println(String.format("%,.2f", amountPaid));
    }
}
```

TECHNICAL STUFF

Why does the title for Listing 10-3 call that code "fake"? (Should the code feel insulted?) Well, the main difference between Listing 10-3 and the inheritance situation in Listings 10-1 and 10-2 is this: A child class can't directly reference the private fields of its parent class. To do anything with the parent class's private fields, the child class has to call the parent class's getter and setter methods. Back in Listing 10-2, calling `setName("Rufus")` would be legal, but the code `name="Rufus"` wouldn't be. If you believe everything you read in Listing 10-3, you'd think that code in the `FullTimeEmployee` class can do `name="Rufus"`. Well, it can't. (My, what a subtle point this is!)

Creating subclasses is habit-forming

After you're accustomed to extending classes, you can get extend-happy. If you created a `FullTimeEmployee` class, you might as well create a `PartTimeEmployee` class, as shown in Listing 10-4.

LISTING 10-4: What Is a PartTimeEmployee?

```
public class PartTimeEmployee extends Employee {
    private double hourlyRate;

    public void setHourlyRate(double hourlyRate) {
        this.hourlyRate = hourlyRate;
    }

    public double getHourlyRate() {
        return hourlyRate;
    }

    public double findPaymentAmount(int hours) {
        return hourlyRate * hours;
    }
}
```

Unlike the `FullTimeEmployee` class, `PartTimeEmployee` has no salary or deduction. Instead, `PartTimeEmployee` has an `hourlyRate` field. (Adding an `hoursWorked` field would also be a possibility. I chose not to do this, figuring that the number of hours a part-time employee works may change from week to week.)

Using Subclasses

The preceding section tells a story about creating subclasses. It's a good story, but it's incomplete. Creating subclasses is fine, but you gain nothing from these subclasses unless you write code to use them.

Listing 10-5 contains the simplest possible example of a program that uses the subclasses `FullTimeEmployee` and `PartTimeEmployee`. (For a look at some more interesting examples, visit this book's website).

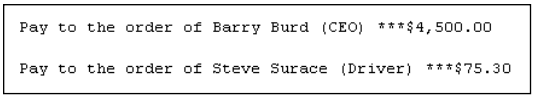
LISTING 10-5: Putting Subclasses to Good Use

```
void main() {

    var ftEmployee = new FullTimeEmployee();

    ftEmployee.setName("Barry Burd");
    ftEmployee.setJobTitle("CEO");
    ftEmployee.setWeeklySalary(5000.00);
    ftEmployee.setBenefitDeduction(500.00);
    ftEmployee.cutCheck(ftEmployee.findPaymentAmount());
    println("");

    var ptEmployee = new PartTimeEmployee();

    ptEmployee.setName("Steve Surace");
    ptEmployee.setJobTitle("Driver");
    ptEmployee.setHourlyRate(7.53);
    ptEmployee.cutCheck(ptEmployee.findPaymentAmount(10));
}
```

Figure 10-2 shows the output of the code in Listing 10-5.

To understand Listing 10-5, you need to keep an eye on three classes: `Employee`, `FullTimeEmployee`, and `PartTimeEmployee`. (For a look at the code that defines these classes, see Listings 10-1, 10-2, and 10-4.)

FIGURE 10-2: I earn a lot more than Steve. Na, na, na!



The first half of Listing 10-5 deals with a full-time employee. Notice the methods that are available for use with the `ftEmployee` variable. You can call `ftEmployee.setWeeklySalary` because `ftEmployee` has type `FullTimeEmployee`. You can also call `ftEmployee.setName` because the `FullTimeEmployee` class extends the `Employee` class.

Because `cutCheck` is declared in the `Employee` class, you can call `ftEmployee.cutCheck`. But you can also call `ftEmployee.findPaymentAmount` because a `findPaymentAmount` method is in the `FullTimeEmployee` class.

Subclasses embody what computer scientists call the *is a* relationship. Every `FullTimeEmployee` instance *is an* `Employee` instance too. To make sure I'm not lying to you, run this code:

```
void main() {
    var ftEmployee = new FullTimeEmployee();
    println(ftEmployee instanceof FullTimeEmployee);
    println(ftEmployee instanceof Employee);
}
```

The output from both `println` calls is the word `true`.

Making types match

Look again at the first half of Listing 10-5. Take special notice of the statement that cuts a check for `ftEmployee`. The statement forms a nice, long chain of values and their types. You can see this by reading the statement from the inside out:

- Method `ftEmployee.findPaymentAmount` is called with an empty parameter list. (Refer to Listing 10-5.) That's good because a full-time employee's `findPaymentAmount` method takes no parameters. (Refer to Listing 10-2.)
- The `findPaymentAmount` method returns a value of type `double`. (Again, refer to Listing 10-2.)
- The `double` value that `ftEmployee.findPaymentAmount` returns is passed to method `ftEmployee.cutCheck`. (Refer to Listing 10-5.) That's good because the `Employee` class's `cutCheck` method takes one parameter of type `double`. (Refer to Listing 10-1.)

For a fanciful graphical illustration of this type-matching business, see Figure 10-3.

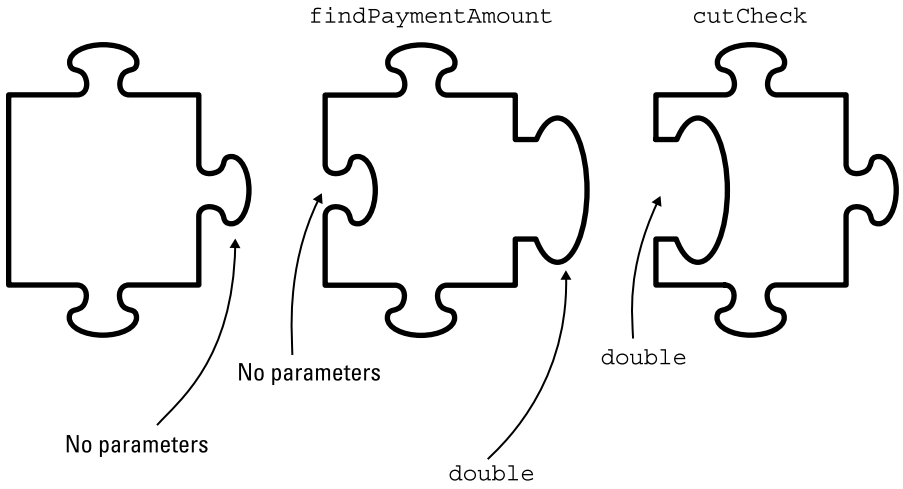


FIGURE 10-3: Matching parameters.

REMEMBER

Always feed a method the value types that it wants in its parameter list.

The second half of the story

In the second half of Listing 10-5, the code creates an object of type `PartTimeEmployee`. A variable of type `PartTimeEmployee` can do some of the same things a `FullTimeEmployee` variable can do. But the `PartTimeEmployee` class doesn't have the `setWeeklySalary` and `setBenefitDeduction` methods. Instead, the `PartTimeEmployee` class has the `setHourlyRate` method. (See Listing 10-4.) So, in Listing 10-5 the next-to-last line is a call to the `setHourlyRate` method.

The last line of Listing 10-5 is by far the most interesting. On that line, the code hands the number `10` (the number of hours worked) to the `findPaymentAmount` method. Compare this with the earlier call to `findPaymentAmount` — the call for the full-time employee in the first half of Listing 10-5. Between the two subclasses, `FullTimeEmployee` and `PartTimeEmployee`, are two different `findPaymentAmount` methods. The two methods have two different kinds of parameter lists:

- The `FullTimeEmployee` class's `findPaymentAmount` method takes no parameters (refer to Listing 10-2).
- The `PartTimeEmployee` class's `findPaymentAmount` method takes one `int` parameter (refer to Listing 10-4).

This is par for the course. Finding the payment amount for a part-time employee isn't the same as finding the payment amount for a full-time employee. A part-time employee's pay changes each week, depending on the number of hours the employee works in a week. The full-time employee's pay stays the same each week. So, the `FullTimeEmployee` and `PartTimeEmployee` classes both have `findPaymentAmount` methods, but each class's method works quite differently.

TRY IT OUT

Yes, I have some things for you to try:

BUY OR RENT

In Chapter 9, you create instances of your own `PlaceToLive` class and display information about those instances. Create two subclasses of your `PlaceToLive` class: a `House` class and an `Apartment` class. Each `House` object has a mortgage cost (a monthly amount) and a property tax cost (a yearly amount). Each `Apartment` object has a rental cost (a monthly amount).

A main method creates some houses and some apartments. For each house or apartment, your main method displays the total cost per square foot (or square meter) and the total cost per bedroom, both calculated monthly.

TAX BREAKS

In Chapter 7, you create an `Organization` class. Each instance of your `Organization` class has a name, an annual revenue amount, and a `boolean` value indicating whether the organization is or is not a profit-making organization.

Create a new `Organization2` class. Each instance of this new class has only a name and an annual revenue amount. Create two subclasses: Call them the `ProfitMakingOrganization` class and a `NonProfitOrganization` class. A profit-making organization pays 10 percent of its revenue in tax, but a nonprofit organization pays only 2 percent of its revenue in tax.

Make a main method that creates `ProfitMakingOrganization` instances as well as `NonProfitOrganization` instances while also displaying information about each instance, including the amount of tax the organization pays.

Changing the Payments for Only Some of the Employees

Wouldn't you know it! Some knucklehead in the human resources department offered double pay for overtime to one of your part-time employees. Now word is getting around, and some of the other part-timers want double pay for their overtime work. If this keeps up, you'll end up in the poorhouse, so you need to send out a memo to all the part-time employees, explaining why earning more money is not to their benefit.

In the meantime, you have two kinds of part-time employees — the ones who receive double-pay-for-overtime hours and the ones who don't — so you need to modify your payroll software. What are your options?

- Well, you can dig right into the `PartTimeEmployee` class code, make a few changes, and hope for the best. (Not a good idea!)
- You can follow the previous section's advice and create a subclass of the existing `PartTimeEmployee` class. "But wait," you say. "The existing `PartTimeEmployee` class already has a `findPaymentAmount` method. Do I need some tricky way of bypassing this existing `findPaymentAmount` method for each double-pay-for-overtime employee?"

  At this point, you can thank your lucky stars that you're doing object-oriented programming in Java. With object-oriented programming, you can create a subclass that overrides the functionality of its parent class. Listing 10-6 has just such a subclass.

LISTING 10-6: Yet Another Subclass

```
public class PartTimeWithOver extends PartTimeEmployee {

    @Override
    public double findPaymentAmount(int hours) {

        if (hours <= 40) {
            return getHourlyRate() * hours;
        } else {
            return getHourlyRate() * 40 + getHourlyRate() * 2 * (hours - 40);
        }
    }
}
```

Figure 10-4 shows the relationship between the code in Listing 10-6 and other pieces of code in this chapter. In particular, `PartTimeWithOver` is a subclass of a subclass. In object-oriented programming, a chain of this kind is not the least bit unusual. In fact, as subclasses go, this chain is rather short.

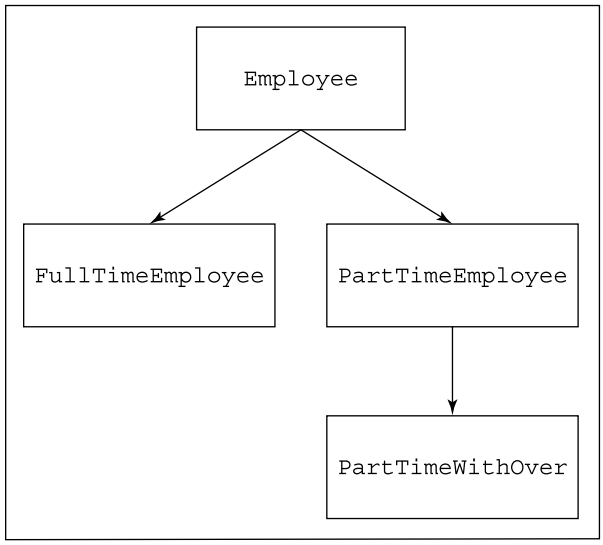


FIGURE 10-4:
A tree of classes.

The `PartTimeWithOver` class extends the `PartTimeEmployee` class, but the `PartTimeWithOver` picks and chooses what it wants to inherit from the `PartTimeEmployee` class. Because `PartTimeWithOver` has its own declaration for the `findPaymentAmount` method, the `PartTimeWithOver` class doesn't inherit a `findPaymentAmount` method from its parent. (See Figure 10-5.)

According to the official terminology, the `PartTimeWithOver` class *overrides* its parent class's `findPaymentAmount` method. If you create an object from the `PartTimeWithOver` class, that object has the `name`, `jobTitle`, `hourlyRate`, and `cutCheck` of the `PartTimeEmployee` class, but the object has the `findPaymentAmount` method that's defined in Listing 10-6.

A Java annotation

The word `@Override` in Listing 10-6 is an example of an *annotation*. A Java annotation tells your computer something about your code. In particular, the `@Override` annotation in Listing 10-6 tells the Java compiler to be on the lookout for a common coding error. The annotation says, "Make sure that the method immediately following this annotation has the same stuff (the same name, the same parameters, and so on) as one of the methods in the superclass. If not, then display an error message."

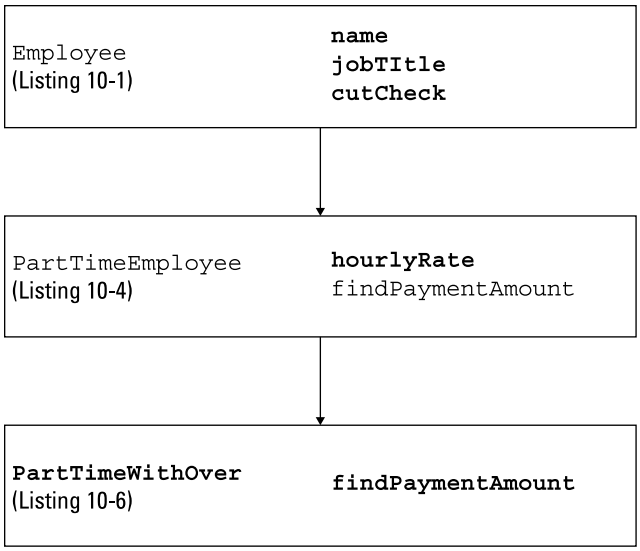


FIGURE 10-5: Method `findPaymentAmount` isn't inherited.

So, in Listing 10-6, if I accidentally type

```
@Override
public double findPaymentAmount(double hours) {
```

with `double hours` instead of `int hours`, the compiler reminds me that my new `findPaymentAmount` method doesn't really override anything that's in Listing 10-4.

Java has other kinds of annotations. For a list, visit `https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html`.

TECHNICAL STUFF

Java's `@Override` annotation is optional. If you remove the word `@Override` from Listing 10-6, your code still runs correctly. But the `@Override` annotation gives your code some added safety. With `@Override`, the compiler checks to make sure that you're doing something you intend to do (namely, overriding one of the superclass's methods). And with apologies to George Orwell, some types of annotations are less optional than others. You can omit certain annotations from your code only if you're willing to replace the annotation with lots and lots of unannotated Java code.

Using methods from classes and subclasses

If you need clarification on this notion of overriding a method, look at the code in Listing 10-7. A run of that code is shown in Figure 10-6.

```
Pay to the order of Barry Burd (CEO) ***$4,500.00
Pay to the order of Chris Apelian (Computer Book Author) ***$376.50
Pay to the order of Steve Surace (Driver) ***$527.10
```

FIGURE 10-6: Running the code of Listing 10-7.

LISTING 10-7: **Paying Three Kinds of Employees**

```
void main() {
    var ftEmployee = new FullTimeEmployee();

    ftEmployee.setName("Barry Burd");
    ftEmployee.setJobTitle("CEO");
    ftEmployee.setWeeklySalary(5000.00);
    ftEmployee.setBenefitDeduction(500.00);
    ftEmployee.cutCheck(ftEmployee.findPaymentAmount());

    var ptEmployee = new PartTimeEmployee();

    ptEmployee.setName("Chris Apelian");
    ptEmployee.setJobTitle("Computer Book Author");
    ptEmployee.setHourlyRate(7.53);
    ptEmployee.cutCheck(ptEmployee.findPaymentAmount(50));

    var ptoEmployee = new PartTimeWithOver();

    ptoEmployee.setName("Steve Surace");
    ptoEmployee.setJobTitle("Driver");
    ptoEmployee.setHourlyRate(7.53);
    ptoEmployee.cutCheck(ptoEmployee.findPaymentAmount(55));
}
```

The code in Listing 10-7 writes checks to three employees. The first employee is a full-timer. The second is a part-time employee who hasn't yet gotten wind of the overtime payment scheme. The third employee knows about the overtime payment scheme and demands a fair wage.

With the subclasses, all three of these employees coexist in Listing 10-7. Sure, one subclass extends the old `PartTimeEmployee` class, but that doesn't mean you can't create an object from the `PartTimeEmployee` class. In fact, Java is smart about this. Listing 10-7 has three calls to the `findPaymentAmount` method, and each call reaches out to a different version of the method:

- In the first call, `ftEmployee.findPaymentAmount`, the `ftEmployee` variable is an instance of the `FullTimeEmployee` class. So the method that's called is the one in Listing 10-2.

- In the second call, `ptEmployee.findPaymentAmount`, the `ptEmployee` variable is an instance of the `PartTimeEmployee` class. So the method that's called is the one in Listing 10-4.
- In the third call, `ptoEmployee.findPaymentAmount`, the `ptoEmployee` variable is an instance of the `PartTimeWithOver` class. So the method that's called is the one in Listing 10-6.

This code is fantastic. Using subclasses makes the code clean and elegant. With all the money that you save on software, you can afford to pay everyone double for overtime hours. (Whether you do that or keep the money for yourself is another story.)

TRY IT OUT

Here are some things for you to try.

PAY MORE AND MORE

In previous sections, you create `House` and `Apartment` subclasses of your `PlaceToLive` class. Create an `ApartmentWithFees` subclass of your `Apartment` class. In addition to the monthly rental price, someone living in an `ApartmentWithFees` pays a fixed amount every quarter (every three months). Create a main method that displays the monthly cost of living in a `House` instance, an `Apartment` instance, and an `ApartmentWithFees` instance.

VIRTUAL METHODS

Create a project with four Java source files. Each file defines a particular Java class:

```
void main() {
    MyThing myThing, myThing2;

    myThing = new MySubThing();
    myThing2 = new MyOtherThing();

    myThing.value = 7;
    myThing2.value = 44;
    myThing.display();
    myThing2.display();
}

public class MyThing {
    int value;
```

```
    public void display() {
        System.out.println("In MyThing, value is " + value);
    }
}

public class MySubThing extends MyThing {

    @Override
    public void display() {
        System.out.println("In MySUBThing, value is " + value);
    }
}

public class MyOtherThing extends MyThing {

    @Override
    public void display() {
        System.out.println("In MyOTHERThing, value is " + value);
    }
}
```

What output do you see when you run the `Main.java` file's code? What does this output tell you about variable declarations and method calling in Java?

THE WAITING GAME

In Chapter 8, you create your own `AirplaneFlight` class. Create a subclass that has a method named `duration`. The `duration` method, which has no parameters, returns the amount of time between the flight's departure time and arrival time.

To find the number of hours between two `LocalTime` objects (such as `twoFifteen` and `currentTime`), execute

```
long hours = ChronoUnit.HOURS.between(twoFifteen, currentTime);
```

To find the number of minutes between two `LocalTime` objects (such as `twoFifteen` and `currentTime`), execute

```
long minutes = ChronoUnit.MINUTES.between(twoFifteen, currentTime);
```

Yet Another switch

From one version of Java to the next, the `switch` structure becomes more and more versatile. With Java 23, a `switch` does almost anything you want it to do. The folks at Wiley don't know it yet, but someday, they'll have to publish a book titled *Java's switch Statements and Expressions For Dummies*.

Listing 10-8 demonstrates the `switch` structure's *pattern-matching* feature.

LISTING 10-8: What Will They Think of Next?

```
void main() {
    var ptEmployee = new PartTimeEmployee();
    ptEmployee.setName("Chris Apelian");
    ptEmployee.setJobTitle("Computer Book Author");
    ptEmployee.setHourlyRate(7.53);
    payEmployee(ptEmployee);

    var ptoEmployee = new PartTimeWithOver();
    ptoEmployee.setName("Steve Surace");
    ptoEmployee.setJobTitle("Driver");
    ptoEmployee.setHourlyRate(7.53);
    payEmployee(ptoEmployee);

    var employee = new Employee();
    payEmployee(employee);
}

void payEmployee(Employee employee) {
    employee.cutCheck(switch (employee) {
        case FullTimeEmployee fte -> fte.findPaymentAmount();
        case PartTimeWithOver pto -> pto.findPaymentAmount(55);
        case PartTimeEmployee pte -> pte.findPaymentAmount(50);
        case Employee e -> 0.00;
    });
}
```

When you run the code in Listing 10-8, you get the output shown in Figure 10-7. In Listing 10-8, the `employee` object has no name and no job title. So, on the last line in Figure 10-7, that person's name and title are both `null`. In Java, `null` means "no value."

```
Pay to the order of Chris Apelian (Computer Book Author) ***$376.50
Pay to the order of Steve Surace (Driver) ***$527.10
Pay to the order of null (null) ***$0.00
```

FIGURE 10-7: The `switch` statement does what it's supposed to do.

The main method in Listing 10-8 calls `payEmployee` three times. Each time, the `payEmployee` method dives into a `switch` expression. Like the `switch` structures in earlier examples, this `switch` divides into cases. But unlike those earlier `switch` structures, this expression's cases are the names of classes. The `switch` expression asks, "What subclass does the `employee` parameter belong to?" Depending on the answer, the `switch` expression calls one of the subclasses' `findPaymentAmount` methods (or gives up and returns `0.00`).

If you mess with the cases in Listing 10-8, you may run into trouble. For example, try changing the order of the middle two cases, like this:

```
// Bad code with cases reversed!
case PartTimeEmployee pte -> pte.findPaymentAmount(50);
case PartTimeWithOver pto -> pto.findPaymentAmount(55);
```

Java absolutely despises this code, and lets you know by displaying a harsh message. "Label is dominated by a preceding case label," says Java. Why does Java say this? I'm glad you asked.

Think about the `ptoEmployee` object in Listing 10-8. It's an instance of the `PartTimeWithOver` class. But the `PartTimeWithOver` class is a subclass of `PartTimeEmployee`. So `ptoEmployee` is also an instance of the `PartTimeEmployee` class. In fact, `ptoEmployee` is an instance of all three classes — `PartTimeWithOver`, `PartTimeEmployee`, and `Employee`. Java's `switch` expression looks for a match from the topmost case downward. (See Figure 10-8.) So, in the Cases Reversed code, the `ptoEmployee` object matches the `PartTimeEmployee` case and never reaches the `PartTimeWithOver` case. In fact, nothing ever reaches the `PartTimeWithOver` case. The `PartTimeWithOver` case is useless, so Java refuses to accept the code.

Here's another thing you can do to ruin the case expression in Listing 10-8. Try removing the last `Employee e` case. Java's `switch` expression must deliver a value no matter what kind of `Employee` instance you feed to it. But the `employee` object near the end of the listing's main method matches none of the first three case clauses. So, without the `Employee e` case, Java makes an unpleasant fuss.

The cases in a `switch` expression of this kind must cover all possibilities. Even if you remove the last two statements in the `main` method in Listing 10-8, Java insists on having a `case` clause that can catch a `new Employee()` object.

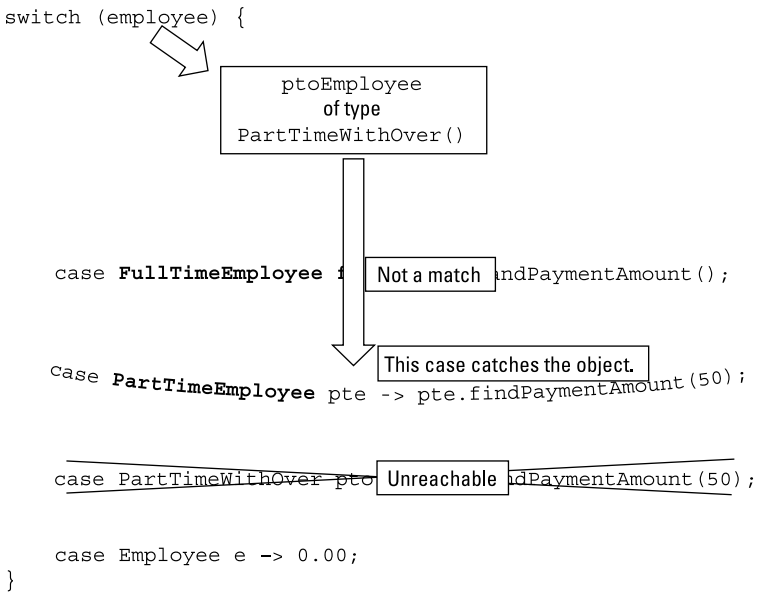


FIGURE 10-8: Nothing good happens.

TIP

Rather than list every possibility, you can get lazy and end a `switch` expression with a `default` clause. Just don't put both the `Employee` case and a `default` clause in Listing 10-8. Java would rebel against that, too.

Doing Something about the Weather

Here's some feedback I got from a reader:

> Hi Barry,
>
> In Chapter 8, the way you display a temperature bothers me. Instead of displaying `70.00° FAHRENHEIT`, you should display `70 °F`. What gives with your crazy temperature notation?
>
> All the best,
>
> Persnickety Percy

And I replied thusly:

> Hi Persnickety,
>
> To display temperatures your way, create a subclass of my `Temperature` class. You can find the code in Listing 10-9.
>
> Regards,
>
> Barry

LISTING 10-9: **Persnickety Percy Gets His Way**

```
public class TemperatureForPercy extends Temperature {

    public TemperatureForPercy() {
        super(); // This super call is optional. (See this
                 // chapter's "Constructor madness" section.)
    }

    public TemperatureForPercy(double number) {
        super(number);
    }

    public TemperatureForPercy(TempScale scale) {
        super(scale);
    }

    public TemperatureForPercy(double number, TempScale scale) {
        super(number, scale);
    }

    @Override
    public String toString() {
        return String.format("%2.0f \u00B0", getNumber()) +
                switch (getScale()) {
                    case CELSIUS -> "C";
                    case FAHRENHEIT -> "F";
                    case KELVIN -> "K";
                    case RANKINE -> "R";

                };
    }

    public double toCelsius() {
        return switch (getScale()) {
            case CELSIUS -> getNumber();
            case FAHRENHEIT -> (getNumber()- 32) * 5.0 / 9.0;
            case KELVIN -> getNumber()- 273.15;
            case RANKINE -> (getNumber()- 491.67) * 5.0 / 9.0;
        };
    }
}
```

In the `toString` and `toCelsius` methods of Listing 10-9, notice the calls to the `Temperature` class's `getNumber` and `getScale` methods. Why do I do this? Well, inside the `TemperatureForPercy` class's code, any direct references to the `number` and `scale` fields would generate error messages. It's true that every `TemperatureForPercy` instance has its own `number` and `scale` fields. (After all, `TemperatureForPercy` is a subclass of the `Temperature` class, and the code for the `Temperature` class defines the `number` and `scale` fields.) But because `number` and `scale` are declared to be private inside the `Temperature` class, only code that's right inside the `Temperature` class can directly use these fields.

WARNING

Don't put additional declarations of the `number` and `scale` fields inside the `TemperatureForPercy` class's code. If you do, you inadvertently create four different variables (two called `number` and another two called `scale`). You'll assign values to one pair of variables. Then you'll be shocked that when you display the other pair of variables, those values seem to have disappeared.

REMEMBER

When an object's code contains a call to one of the object's own methods, you don't need to preface the call with a dot. For instance, in the `toString` method of Listing 10-9, the object calls its own methods with `getNumber()` and `getScale()`, not with `someObject.getNumber()` and `somethingOrOther.getScale()`. If going dotless makes you queasy, you can compensate by taking advantage of yet another use for the `this` keyword: Just write `this.getNumber()` and `this.getScale()` in the `toString` method in Listing 10-9.

Constructors for subclasses

By far, the biggest news in Listing 10-9 is the way the code declares constructors. The `TemperatureForPercy` class has four of its own constructors. If you've gotten in gear thinking about subclass inheritance, you may wonder why these constructor declarations are necessary. Doesn't `TemperatureForPercy` inherit the parent `Temperature` class's constructors? No, subclasses don't inherit constructors.

REMEMBER

Subclasses don't inherit constructors.

That's right. Subclasses don't inherit constructors. The default constructors in Chapter 8 may look as if they're inherited, but they're not. In general, when you define a subclass, you declare new constructors to go with the subclass.

So the code in Listing 10-9 has four constructors. Each constructor has the name `TemperatureForPercy`, and each constructor has its own uniquely identifiable parameter list. That's the boring part. The interesting part is that each constructor makes a call to something named `super`, which is a Java keyword.

In Listing 10-9, `super` stands for a constructor in the parent class:

- The statement `super()` in Listing 10-9 calls the parameterless `Temperature()` constructor that's in Listing 8-2. (Yes, you'll need to flip back to Chapter 8 to find it.) That parameterless constructor assigns `0.0` to the `number` field and `TempScale.FAHRENHEIT` to the `scale` field.
- The statement `super(number, scale)` in Listing 10-9 calls the constructor `Temperature(double number, TempScale scale)` that's in Listing 8-2. In turn, the constructor assigns values to the `number` and `scale` fields.
- In a similar way, the statements `super(number)` and `super(scale)` in Listing 10-9 call constructors from Listing 8-2.

The computer decides which of the `Temperature` class's constructors is being called by looking at the parameter list after the word `super`. For instance, when the computer executes

```
super(number, scale);
```

from Listing 10-9, the computer says to itself, "The `number` and `scale` fields in parentheses have types `double` and `TempScale`. And only one of the `Temperature` constructors in Listing 8-2 has two parameters with types `double` and `TempScale`. The constructor's header looks like this:

```
public Temperature(double number, TempScale scale)
```

"So I guess I'll execute the statements inside that particular constructor."

Using all this stuff

In Listing 10-9, I define what it means to be in the `TemperatureForPercy` class. Now it's time to put this `TemperatureForPercy` class to good use. Listing 10-10 has code that uses `TemperatureForPercy`.

LISTING 10-10: Using Percy's Class

```
void main() {
    println(new TemperatureForPercy(2.73, TempScale.KELVIN));
    println(new TemperatureForPercy(TempScale.CELSIUS));
    println(new TemperatureForPercy(32.0));

    var temperature = new TemperatureForPercy();
    temperature.setNumber(70);
    println(temperature);
}
```

Listing 10-10 creates instances of the `TemperatureForPercy` class, so the listing's `println` calls access the `toString` method in the `TemperatureForPercy` class, not the parent `Temperature` class.

Figure 10-9 shows a run of the code in Listing 10-10.

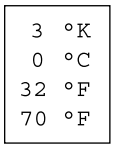


FIGURE 10-9: At last! Percy is happy.

Read the comment after the first `super` call in Listing 10-9. Try removing that `super` call and notice that you still get the output shown in Figure 10-9. What does this tell you? It tells you that, under certain circumstances, Java adds a parameterless `super()` call to your constructor.

And what are those "certain circumstances"? The next section reveals the answer.

Constructor madness

The big message in the previous section is that subclasses don't inherit constructors. When you combine this with some other rules concerning constructors, you get some surprising outcomes. Here are some facts about constructors:

1. A subclass doesn't inherit any constructors from its parent class.
2. When you define a class, and you don't explicitly type the code for any constructors, Java generates a parameterless default constructor for you.

   This rule goes both ways. If you explicitly type the code for a constructor, Java *doesn't* generate a default constructor for you.

3. When a class contains a constructor, and that constructor doesn't contain a statement of this kind:

```
super(some parameters or no parameters);
```

then Java generates a parameterless `super()` call for you. In other words, Java calls the parent class's parameterless constructor on your behalf.

To see these rules in action, consider the code in Listings 10-11 through 10-13.

LISTING 10-11: Getting Help in the Kitchen

```
public class KitchenItem {
    public String purpose;

    public KitchenItem() {
        System.out.println("""
                You've called the parameterless
                KitchenItem constructor.""");
    }
}
```

LISTING 10-12: Don't Eat with Your Hands

```
enum Material {METAL, PLASTIC, WOOD}

public class Utensil extends KitchenItem {
    public Material material;

    public Utensil(String purpose, Material material) {
        this.purpose = purpose;
        this.material = material;
    }

    // Some code is missing.
}
```

LISTING 10-13: You May Have to Google the Word *Tine*

```
public class Fork extends Utensil {
    public int numberOfTines;
}
```

If you type all this code, Listing 10-13 gives you an error message such as, `There is no default constructor available in 'Utensil'`. Why is Java complaining to you? Have you done something wrong? Figure 10-10 illustrates the situation.

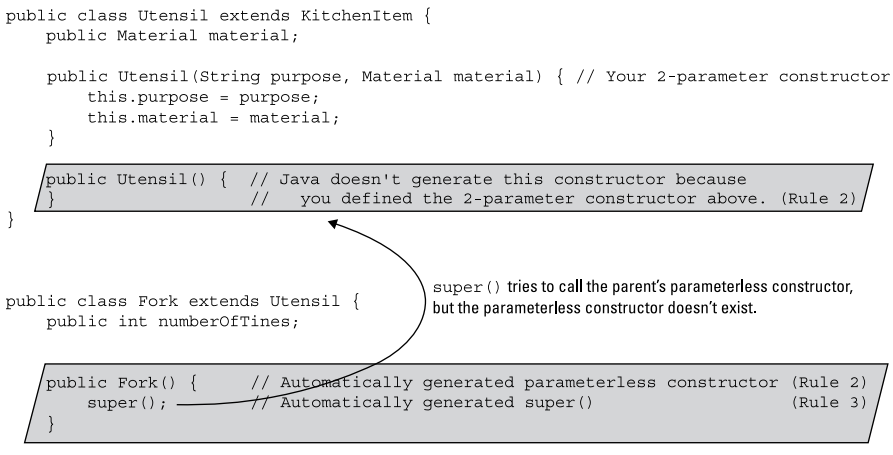

public class Utensil extends KitchenItem {
public Material material;
public Utensil(String purpose, Material material) { // Your 2-parameter constructor
this.purpose = purpose;
this.material = material;
}
public Utensil() { // Java doesn't generate this constructor because
} // you defined the 2-parameter constructor above. (Rule 2)
}
super() tries to call the parent's parameterless constructor, but the parameterless constructor doesn't exist.
public class Fork extends Utensil {
public int numberOfTines;
public Fork() { // Automatically generated parameterless constructor (Rule 2)
super(); // Automatically generated super() (Rule 3)
}
}

FIGURE 10-10: Implicitly generated code makes trouble for you.

With Rules 2 and 3, an implicit default constructor inside the `Fork` class tries to call a parameterless `Utensil` constructor. But Rule 2 tells Java not to create a parameterless `Utensil` constructor. So the `super()` in the `Fork` class's default constructor has nothing to call. Java recognizes this difficulty and insists that you repair your code.

You can fix this by adding an explicit parameterless constructor to the `Utensil` class in Listing 10-12:

```
public class Utensil extends KitchenItem {
    public Material material;

    public Utensil(String purpose, Material material) {
        this.purpose = purpose;
        this.material = material;
    }

    public Utensil() {
    }
}
```

Figure 10-11 shows you what happens when you add this constructor.

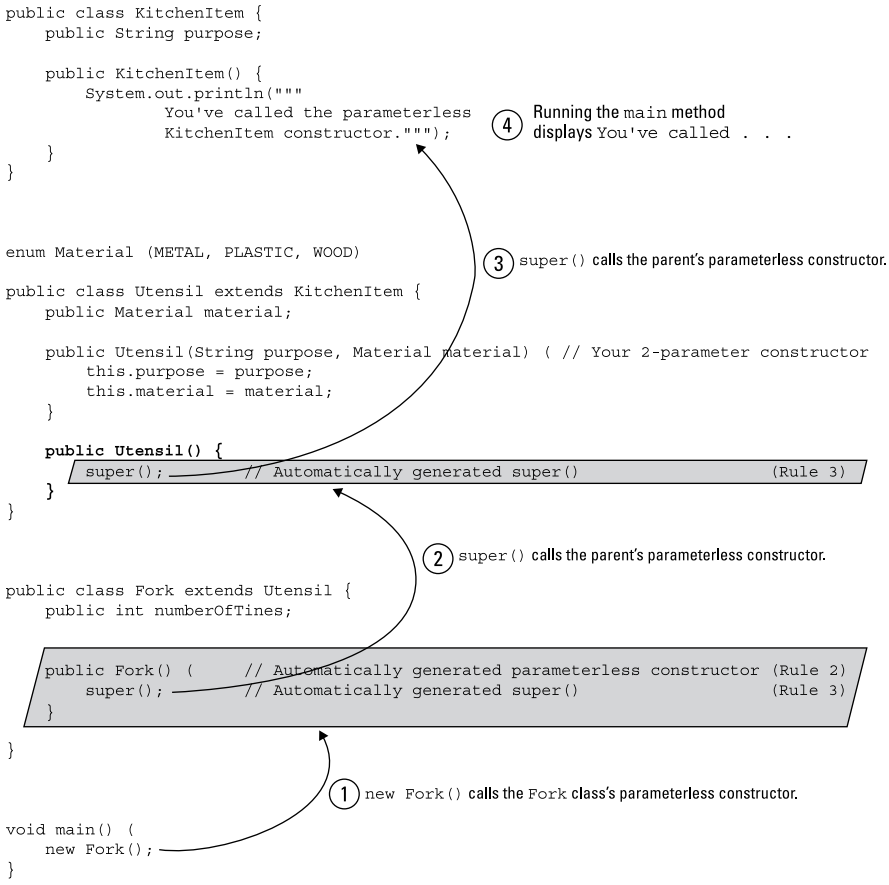

public class KitchenItem {
public String purpose;
public KitchenItem() {
System.out.println("""
You've called the parameterless
KitchenItem constructor.""");
}
}
4 Running the main method displays You've called . . .
enum Material (METAL, PLASTIC, WOOD)
public class Utensil extends KitchenItem {
public Material material;
public Utensil(String purpose, Material material) (// Your 2-parameter constructor
this.purpose = purpose;
this.material = material;
}
public Utensil() {
super(); // Automatically generated super() (Rule 3)
}
}
3 super() calls the parent's parameterless constructor.
2 super() calls the parent's parameterless constructor.
public class Fork extends Utensil {
public int numberOfTines;
public Fork() (// Automatically generated parameterless constructor (Rule 2)
super(); // Automatically generated super() (Rule 3)
}
}
1 new Fork() calls the Fork class's parameterless constructor.
void main() (
new Fork();
}

FIGURE 10-11: One constructor call after another.

In Figure 10-11, a call to the `Fork` class's default constructor sets off a chain of calls that ends with a call to the `KitchenItem` constructor. Normally, you don't put `println` calls of any kind inside a constructor declaration. But in this example, the `println` I've added helps to convince you that Java obeys Rules 2 and 3.

TRY IT OUT

Here's a neat little program for you to write:

A CONVERSION DIVERSION

You may have noticed that the `TemperatureForPercy` class in Listing 10-9 has a `toCelsius` method. Create a subclass of `TemperatureForPercy` named

`TemperatureCelsius`. The `TemperatureCelsius` class has only three constructors. When you run this main method:

```
void main() {
    println(new TemperatureCelsius());

    println(new TemperatureCelsius(100.00));

    var temp = new TemperatureForPercy(70.0, TempScale.FAHRENHEIT);
    println(new TemperatureCelsius(temp));
}
```

the output is

```
0 °C
100 °C
21 °C
```

4 Smart Java Techniques

IN THIS PART . . .

Decide where declarations belong in your Java program.

Deal with bunches of things (bunches of rooms, bunches of sales, and even bunches of bunches).

Keep cool even when something goes wrong with your program.

Fully embrace Java's object-oriented features.

Create a windowed app and respond to mouse-clicks.

IN THIS CHAPTER

- Making something belong to an entire class
- Putting variables inside and outside methods
- Improving your batting average

Chapter 11

Putting Variables and Methods Where They Belong

Our family's neighborhood elementary school has a yearly tradition: Each spring, the students in fifth grade take a daylong field trip to Philadelphia. When my son was in fifth grade, I went along on the trip to help supervise 4 of the 20 children. While the 4 kids and I were touring the city's historical sites, I misplaced the preplanned itinerary. I didn't know where we were supposed to go next. So, what could I do? I did what any good Philadelphia native would do — I ignored the itinerary and took the kids to the nearest cheap restaurant. I treated them all to Philadelphia's signature dish — the fat-laden, greasy, Philadelphia cheesesteak.

What could this story possibly have to do with Java? That's simple. This chapter is about access — how one class's code may gain access to another class's code. My Philadelphia anecdote is about access, too. Shortly after the school trip, my son's fifth-grade teacher got wind of my shenanigans during the tour. Later, when my daughter was in fifth grade, the teacher said that I wasn't welcome to take part in that year's junket to Philadelphia. The teacher had revoked my access to those Philadelphia cheesesteaks.

Making a Package

Today, our local baseball team plays against the neighboring town's team. Our team is the Hankies; the other town's team is the Socks. As the team's official recordkeeper, I've created code to keep track of the players' names and batting averages. (See Listing 11-1.)

LISTING 11-1: The Player Record Class

```
package com.example.baseball;

public record Player(String name, double average) {
}
```

Wow! That's a tiny piece of code! (If you have doubts about what's going on in Listing 11-1, refer to the section on record classes in Chapter 8.) To sum it up, each `Player` instance has a `name` field and an `average` field. There's only one way to create a `Player` instance, and that's by calling `new Player(`*`a_name, an_average`*`)`. Once you've created a player, you can't change that player's name or average. To get a particular player's name and average, you call the record class's `name()` and `average()` methods. That's all there is to it.

For comparison, the `Player` class in previous editions of *Java For Dummies* looked something like this:

```
public class Player {
    String name;
    double average;

    public Player(String name, double average) {
        this.name = name;
        this.average = average;
    }
}
```

This older code did mostly what the code in Listing 11-1 does.

Package deal

The `Player` class in Listing 11-1 belongs to a package that I named `com.example.baseball`. The first line in Listing 11-1 makes it so. That first line is called a

package declaration. That line says, "This file is one of possibly many files that are grouped inside a package named `com.example.baseball`."

In Java, every class belongs to a package. Imagine a file named `Hello.java` containing only this text:

```
void main() {
    println("Hello");
}
```

Because of the file's name, this `main` method belongs to an implicit class named `Hello`. And that `Hello` class belongs to an implicit package. Java developers call it an *unnamed package*.

None of the listings in previous chapters starts with a package declaration, so each of those listings belongs to an unnamed package. But professional Java programmers avoid using unnamed packages. So, in Listing 11-1, I decided to straighten up and start flying right.

The Java API defines thousands of classes, each belonging to one of several hundred packages. When you write `import java.util.Scanner`, you're referring to a class named `Scanner`, which belongs to the package `java.util`.

In the Java world, programmers customarily give packages long, dot-filled names. For example, I've registered the domain name *`allmycode.com`*, so I may decide to name a package `com.allmycode.dummiesframe`. In truth, I can name a package almost anything I want — like `com.example.baseball`, `fish.and.chips`, or simply `hamburger`. In any case, informative package names like `com.allmycode.dummiesframe` are better than silly names like `midnight.in.akron` or `do.not.run.this.code`.

TIP

In this book, I start most of my package names with `com.example`. The Internet Assigned Numbers Authority has set aside the *`example.com`* domain for anyone to use in writing documentation, illustrating concepts, describing ideas, and so on.

How many classes can you define? The `Player` class in Listing 11-1 is one of them. You can also create a `Game` class, a `Season` class, and classes named `Inning`, `Bat`, `Ball`, `Strike`, `HomeRun`, `Foul`, `Base`, `Pitcher`, `Catcher`, `Umpire`, `Manager`, `Infielder`, `Outfielder`, `DoublePlay`, and many more. Of course, you may want to separate these classes from your other classes named `Song`, `Novel`, `Film`, `Painting`, and `Video`. So you might create two different packages: one named `com.example.baseball` and another named `com.example.arts`.

You can start any Java file with a package declaration, but you have to remember a few things:

- **A Java file's package name dictates the names of some folders and subfolders.**

  The code in Listing 11-1 begins with `package com.example.baseball`. So your hard drive must have a folder named `com`, which has a subfolder named `example`, which in turn has its own subfolder named `baseball`. Your `Player.java` file must be inside this `baseball` folder. For example, on my computer, I have a project whose root folder is named `11-01`. My copy of Listing 11-1 is inside the `11-01/src/com/example/baseball` folder.

- **Many IDEs have to be told explicitly to create the needed folders.**

  Before you copy the code from Listing 11-1 into your IDE, you probably have to tell the IDE to create a package named `com.example.baseball`. When you do, your IDE will create the `com/example/baseball` folders for you. Then you can copy the `Player.java` file into the `com/example/baseball` folder.

  How do you tell your IDE to create a package? When you start a new project, your IDE might prompt you for a new package name. The IDE creates the package as part of the new project. When you work on an existing project, creating a package might mean fishing around for New ➪ Package among your IDE's menus. If you can't find a way to do that, look for some hints on this book's website (`http://javafordummies.allmycode.com`).

After all this fuss about packages, you may wonder what difference it makes that a class is in one package or another. For some insight, see this chapter's later section "It's All About Access."

Using the Player class

Listing 11-2 has code that uses the `Player` class — the class that's defined earlier, in Listing 11-1.

LISTING 11-2: Using the Player Class

```
import com.example.baseball.Player;

import java.io.*;
import java.util.Scanner;

public class ShowPlayers {

    public static void main(String[] args) throws IOException {
```

```
        var hankiesData = new Scanner(new File("Hankies.txt"));

        IO.print("Player");
        IO.println(String.format("%45s", "Batting Average"));
        IO.print("------");
        IO.println(String.format("%45s", "---------------"));

        for (int num = 1; num <= 9; num++) {
            var player = new Player(hankiesData.nextLine(),
                    hankiesData.nextDouble());
            hankiesData.nextLine();
            IO.print(String.format("%-40s", player.name()));
            IO.println(String.format("%.3f", player.average()).substring(1));
        }

        hankiesData.close();
    }
}
```

WHAT'S ALL THE FUSS ABOUT?

Would you like to see a simple source file replacement for the code Listing 11-2? Well, here it is:

```
import com.example.baseball.Player;

void main() throws IOException {
    var hankiesData = new Scanner(new File("Hankies.txt"));

    print("Player");
    println(String.format("%45s", "Batting Average"));
    print("------");
    println(String.format("%45s", "---------------"));

    for (int num = 1; num <= 9; num++) {
        var player = new Player(hankiesData.nextLine(),
                hankiesData.nextDouble());
        hankiesData.nextLine();
        print(String.format("%-40s", player.name()));
        println(String.format("%.3f", player.average()).substring(1));
    }

    hankiesData.close();
}
```

(continued)

(continued)

Here's how this simple source file differs from the code in Listing 11-2:

- In this code, you don't explicitly create a class to contain the main method. Instead, Java implicitly defines a containing class for you.
- Implicitly declared classes automatically import everything inside the `java.base` module. In this sidebar's code, you don't have to write import declarations for `java.io` and `java.util` because both of these packages are in the `java.base` module.
- Sorry! You need an explicit import declaration for `com.example.baseball.Player`. The people who decide what's part of the Java API didn't create the `com.example.baseball.Player` class. I did.
- In this sidebar's code, you can skip `public static` and `String[] args`. With their infinite kindness, the lords of Java 23 have deemed these words to be unnecessary.
- In this code, you don't have to preface the `print` and `println` calls with `IO`. When you create an implicitly declared class, Java imagines that you've written static `import` declarations for these methods. (Well, wha' da' ya' know? There's a section titled "All about static import" in this very chapter!)
- With this sidebar's code, you still have to use the `Scanner` class to read values. You can't use the simpler `readln` method because `readln` gets values only from the keyboard. In this example, you read from a disk file.

This sidebar's code has a few brand-new features, and so does Listing 11-2. You can read about these features in the pages that follow.

In a typical Java application, the application's classes belong to several different packages. And the code in those packages depends on code from other organizations' packages. Examples like the ones in this book, with only one or two packages, are rare indeed.

For my baseball example, I put the `Player` class in a package that I name `com.example.baseball` and put the `ShowPlayers` class in Java's ever-present unnamed package. Since the `ShowPlayers` class isn't in the same package as my `Player` class, I start Listing 11-2 with an import declaration. Without the import declaration at the very top of Listing 11-2, I'd have to write

```
var player = new com.example.baseball.Player(hankiesData.nextLine(),
        hankiesData.nextDouble());
```

inside the listing's `for` loop.

In Listing 11-2, notice how `import java.io.*;` ends with an asterisk. The asterisk means "import every class that's in the `java.io` package." Listing 11-2 abbreviates three of those classes: `IOException`, `File`, and `IO`. But, with an asterisk in the import declaration, the listing could abbreviate `BufferedReader`, `FileInputStream`, or any other classes in the `java.io` package.

TECHNICAL STUFF

A Java package can have one or more *subpackages*. For example, the `javax.swing.text` package is for displaying text of any kind in a graphical window, and the `javax.swing.text.html` subpackage is for displaying HTML text in particular. But don't let the name *subpackage* fool you. The relationship between a package and its subpackage is flimsy. If you declare `import javax.swing.text.*`, you can abbreviate the names of classes in the `javax.swing.text` package — classes like `Caret`, `Highlighter`, `Position`, and `Style`. But you can't abbreviate the names of classes in the `javax.swing.text.html` package. For that, you declare `import javax.swing.text.html.*`.

Running the code

To run this program, you need a file containing data on your favorite baseball players. Fortunately, the stuff you download from this book's website comes with a `Hankies.txt` file. (See Figure 11-1.)

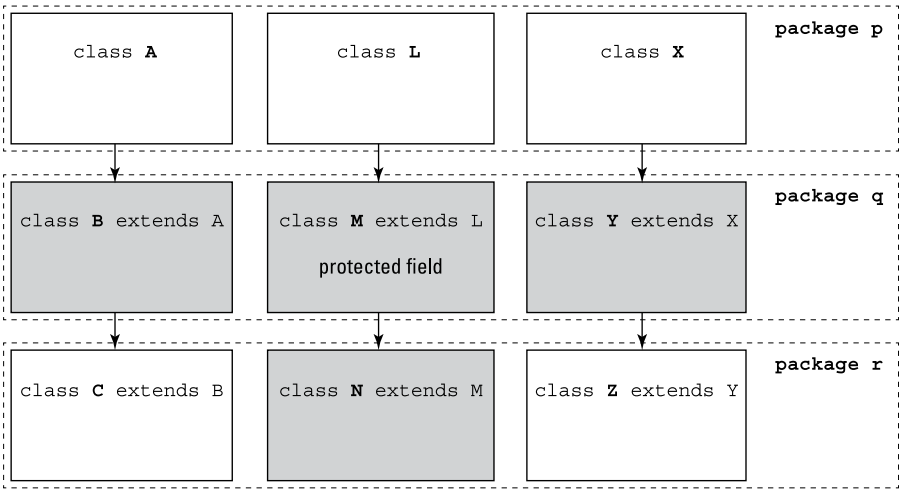


FIGURE 11-1: My `Hankies.txt` file.

For a run of the code in Listing 11-2, see Figure 11-2.

```
Player                                Batting Average
--------                              -------------------
Barry Burd                                  .101
Harriet Ritter                              .200
Weelie J. Katz                              .030
Harry "The Crazyman" Spoonswagler           .124
Filicia "Fishy" Katz                        .075
Mia, Just "Mia"                             .111
Jeremy Flooflong Jones                      .102
I. M. D'Arthur                              .001
Hugh R. DaReader                            .212
```

FIGURE 11-2: What a team!

The program's `Hankies.txt` file can't be in just any old folder on your computer's hard drive. The file has to be in a place where Java can find it, which is most likely your project's root folder. If Java can't find the file, then a run of this section's example gives you an unpleasant `FileNotFoundException` message. For advice on choosing locations for files like `Hankies.txt`, refer to Chapters 6 and 9.

WARNING

You may live in a country where the value of π is approximately 3,14159 (with a comma) instead of 3.14159 (with a period). If you do, the file shown in Figure 11-1 won't work for you. The program will crash with an `InputMismatchException`. To run this section's example, you have to change the periods in the `Hankies.txt` file into commas. Alternatively, you can add a statement such as `Locale.setDefault(Locale.US)` to your code. For details, see Chapter 9.

REMEMBER

For this section's code to work correctly, you must have a line break after the last `.212` in Figure 11-1. For details about line breaks, see Chapter 9.

One class; nine objects

The code in Listing 11-2 calls the `Player` constructor nine times. The result is that the code creates nine instances of the `Player` class. The first time through the loop, the code creates an instance with the name `Barry Burd`. The second time through the loop, the code abandons the `Barry Burd` instance and creates another instance with name `Harriet Ritter`. The third time through, the code abandons poor `Harriet Ritter` and creates an instance for `Weelie J. Katz`. The code has only one instance at a time but, all in all, the code creates nine instances.

Each `Player` instance has its own `name` and `average` fields. Each instance also has its own `Player` constructor and, since it's a record class, its own `name()` and `average()` methods. Look at Figure 11-3 and think of the `Player` class with its nine incarnations.

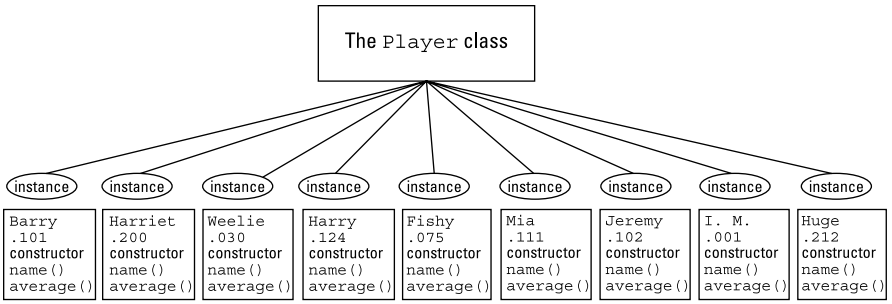

The Player class
instance
Barry .101 constructor name() average()
Harriet .200 constructor name() average()
Weelie .030 constructor name() average()
Harry .124 constructor name() average()
Fishy .075 constructor name() average()
Mia .111 constructor name() average()
Jeremy .102 constructor name() average()
I. M. .001 constructor name() average()
Huge .212 constructor name() average()

FIGURE 11-3: A class and its objects.

Some notes on formatting

Figure 11-4 illustrates the work of the first two printing statements in Listing 11-2. In `"%45s"`, the letter `s` says, "Expect the next parameter to be a string." And the next parameter, `"Batting Average"`, is indeed a string. In that same code, the number `45` says "Use 45 characters to display the string." So Java pads the 15-character `"Batting Average"` string with 30 blank spaces.

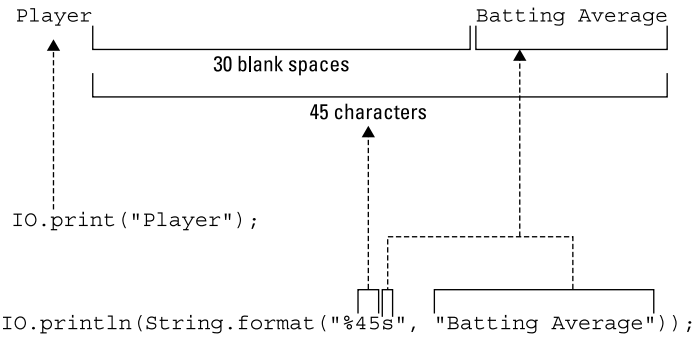


FIGURE 11-4: Printing a header row.

Figure 11-5 shows how a `print` call in Listing 11-2 formats my favorite player's name. In `"%-40s"`, the `40s` says, "Use 40 characters to display the second parameter, which is a string." The minus sign calls for left-justification, so the name is to the left of the 30-character padding.

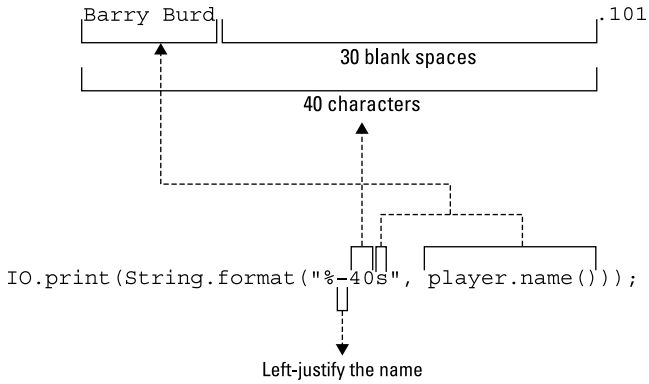


FIGURE 11-5: Printing a player's name.

Figure 11-6 describes the formatting of a player's batting average. In `"%.3f"`, the `.3f` part asks for a number with three digits to the right of the decimal point. For numbers less than 1, Java puts a digit `0` on the left side of the decimal point. That's usually good. (My high-school teacher told me to do that so that the decimal point would be harder to miss.)

But the convention for displaying batting averages is to not have this leading `0`. So, to get rid of the leading `0`, I call Java's `substring` method. In particular,

- The value of `"0.101".substring(0)` is `"0.101"`,
- The value of `"0.101".substring(1)` is `".101"`,
- The value of "0.101".substring(**2**) is "101",

. . . and so on.

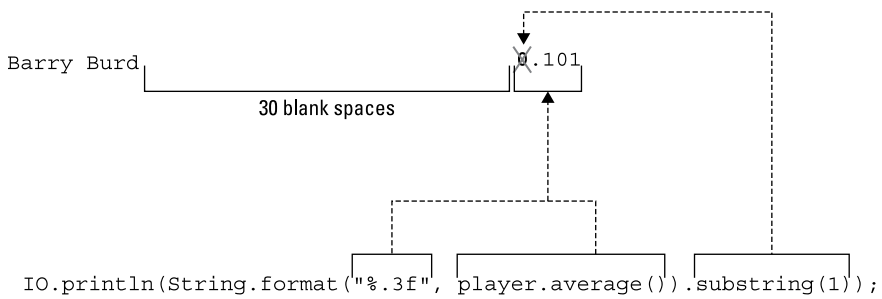


FIGURE 11-6: Printing a player's awful batting average.

TIP

Calling `String.format` isn't the only way to format numbers in Java. Other ways include the `System.out.printf` method, the `DecimalFormat` class, and the `NumberFormat` class's `getCurrencyInstance` method. For more information, see Chapter 12.

TRY IT OUT

Would you like some practice with the material in this section? If so, try this:

COURTING DISASTER

The code in Listing 11-2 reads from a file named `Hankies.txt`. Delete that `Hankies.txt` file from your computer's hard drive, or temporarily move the file to a different directory. Then try to run the program in Listings 11-1 and 11-2. What horrible things happen when you do this?

MEN'S CLOTHING

A line of men's clothing features shirts, pants, jackets, overcoats, neckties, and shoes. Create an `enum` to represent the six kinds of items. Then create a `MensClothingItem` class. Each instance of the class has a kind (one of the six `enum` values) and a name (such as *Casual Summer Design #7*).

Write code to display a table with six rows that describe one complete men's wardrobe.

DEAL ME IN

Create an `enum` to represent the suits in a deck of playing cards (`CLUBS`, `DIAMONDS`, `HEARTS`, and `SPADES`). Create a `PlayingCard` class. Each playing card has a number (from 1 to 13) and a suit. In the numbering scheme, 11 stands for a Jack, 12 stands for a Queen, and 13 stands for a King. Write code that creates several cards and displays them on the screen.

It's All About Access

Many examples in this book throw around the words `public` and `private` as if they're no big deal at all. That's okay because the English-language meanings of the words describe, more or less, what they mean in a Java program. But it's nice to be precise. So this section zooms in on the words `public` and `private`. These words are two of Java's *access modifiers*. The word `protected` is another Java access modifier. And, not using any word (in a place where you can use `public`, `private`, or `protected`) is also a kind of access modifier.

For starters, what does *access* mean? Imagine that you have two fields, two methods, a class with a separate method, or two things of any kind. Does the

name of the first thing mean anything to the second thing? Here are two pieces of code:

```
class First {
    int number = 0;
}

class Second {
    void main() {
        var first = new First();
        first.number = 99;
    }
}
```

Is this code okay? Can the `Second` class's main method refer to the `First` class and to its `number` field? If so, the `Second` class has access to these things.

An *access modifier* helps to determine what kind of access is available. To describe Java's access modifiers, I start with the big picture — modules.

Access for Java modules

There's an old poem that goes something like this:

```
As I was going to high school,
I ran some code with seven modules,
Each module had seven packages,
Each package had seven classes,
Each class had seven members:
Members, classes, packages, and modules,
How many were there going to high school?
```

A *module* is a group of packages. Each module is a world unto itself. If you have two modules and you don't do anything to connect the two modules, then none of the code in either module can make reference to code in the other module.

Imagine that you have two modules: module1 and module2. If you want some code in module2 to make reference to any of the code in module1, then module1 must explicitly *export* the code, and module2 must explicitly *require* the code. You must create two files, both named `module-info.java`, like so:

```
// One module-info.java file:
module module1 {
    exports some_package_in_module1;
}
```

```
// Another module-info.java file:
module module2 {
    requires some_package_in_module1;
}
```

The details can be complicated. But that's okay because most Java programmers never have to create modules. In other words, you're off the hook.

For more information about modules, visit `https://openjdk.org/projects/jigsaw/quick-start`.

From this point on in the chapter, wherever you read about "two packages" or "two classes," imagine that they're both in the same module.

Access modifiers for classes

In Java, the `public` keyword has two slightly different meanings: one meaning for the members of a class and another meaning for the classes themselves. This section deals with the classes themselves.

A class can be either public or nonpublic. If you see something like

```
public class Drawing
```

you're looking at the declaration of a public class. But if you see plain old

```
class OnlyMyDrawing
```

the class that's being declared isn't public.

Public classes

If a class is public, you can refer to the class from anywhere in your code. Of course, some restrictions apply. You must obey all the rules in this chapter's earlier sections "Package deal" and "Access for Java modules." You must also refer to a packaged class properly. For example, in Listing 11-2, you write

```
import com.example.baseball.Player;

        var player = new Player(hankiesData.nextLine(),
                hankiesData.nextDouble());
```

Alternatively, you can do without the import declaration and write

```
        var player = new com.example.baseball.Player(hankiesData.nextLine(),
                hankiesData.nextDouble());
```

One way or another, your code must acknowledge that the `Player` class isn't in the same package as the code in Listing 11-2.

Nonpublic classes

If a class isn't public, you can refer to the class only from code within the class's package.

I tested this rule. I went back to Listing 11-1 and deleted the word *public.* At that point, Listing 11-2 went poof! To be more precise, my IDE displayed these messages for Listing 11-2:

```
'com.example.baseball.Player' is not public in 'com.example.baseball'.
        Cannot be accessed from outside package.
Cannot access 'name()' in 'com.example.baseball.Player.
Cannot access 'average()' in 'com.example.baseball.Player."
```

I guess I got what was coming to me.

Things are never as simple as they seem. The rules I describe in this section apply to almost every class in this book. But Java has fancy things called *sealed classes* and *inner classes*, and these classes follow a different set of rules. Fortunately, a typical novice programmer has little contact with sealed or inner classes. So, for now, you can live quite happily with the rules that I describe in this section.

Enough about access rules for classes! It's time to explore other avenues.

A word or two about method-local variables

Consider this chunk of Java code:

```
public class MyClass {
    int myField;           // This is a field (a member of MyClass).
    // myVariable++;       // Illegal. No access.

    void myMethod() {      // This is a method (another member of MyClass).
        int myVariable;    // This is a method-local variable (NOT a member).
    }
}
```

The comments on the right side of the code tell the whole story. This class has three kinds of members: a field, a method, and a method-local variable. There's not much to say about method-local variables. A method-local variable can't be accessed from any code outside of the method. Period. If you want more details, refer to Chapter 7.

Access for members (fields and methods)

Imagine for a moment that you're a member of a class. In particular, you're a field named `identificationNumber`. From somewhere in the code, you hear the words "Hey, `identificationNumber`! What's your current value?" Your first response should be, "Who wants to know?" Depending on who's asking for the `identificationNumber`, you may or may not be willing to answer.

Members in the same class

Two members in the same class have free and easy access to one another. For example, the following code is right and proper:

```
class MyClass {
    private int x = 7;
    private int y = addOne();

    private int addOne() {
        return x + 1;
    }

    void main() {
        System.out.println(y);
    }
}
```

The output of this silly program is `8`. In this program, every name is available to the rest of the program's code. The `addOne` method uses the value of `x`, the initialization of `y` calls `addOne`, and `main` uses the value of `y`. Almost everything is declared to be private. But because everything is in the same class, the word `private` doesn't matter.

Members in different classes

The ability to access a member of a different class depends on that member's access modifier. Here's the whole story:

PRIVATE ACCESS

If a member is private, only code that's inside the member's class can refer directly to that member's name. Here's some sample code:

```
class SomeClass {
    private int myField = 10;
}

//////////////////////////////

class SomeOtherClass {

    void main() {
        var someObject = new SomeClass();
        System.out.println(someObject.myField);        //This doesn't work.
    }
}
```

For a picture that describes private access, see Figure 11-7.

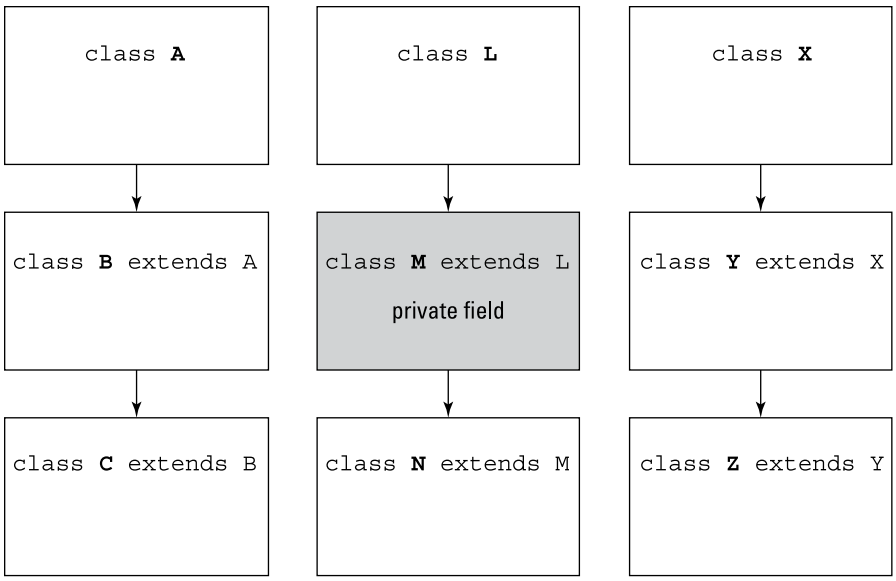


FIGURE 11-7: The range of code in which a private field or method can be used (shaded).

PUBLIC ACCESS

If a member is public, any code can refer directly to that member's name. Here's some sample code:

```
class SomeClass {
    public int myField = 10;
}

//////////////////////////////

class SomeOtherClass {

    void main() {
        var someObject = new SomeClass();

        System.out.println(someObject.myField);       //This works.
    }
}
```

(Remember that both classes have to be in the same module. Otherwise, all bets are off.) For a picture that describes public access, see Figure 11-8.

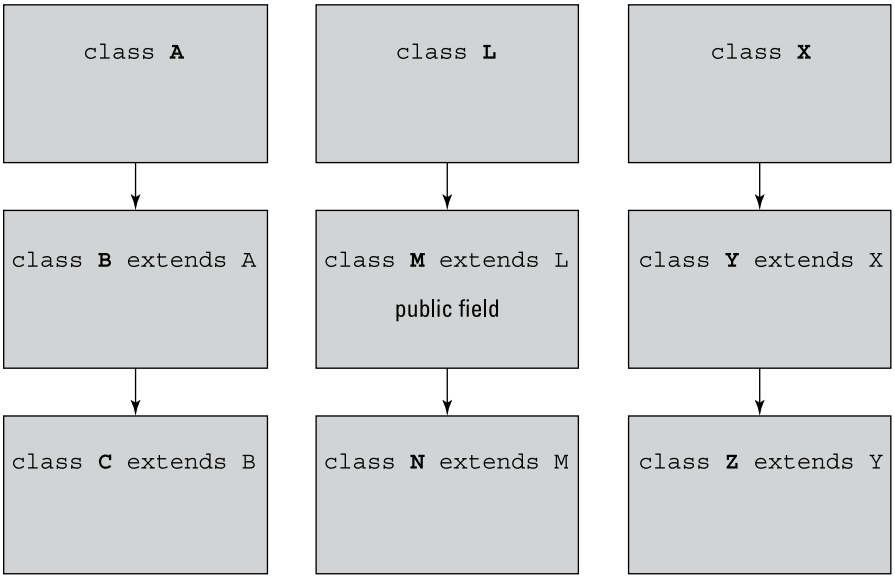


FIGURE 11-8: The range of code in which a public field or method can be used (shaded).

WARNING

Public fields are easy to use and even easier to misuse. The best way to engineer your code is to make access to each field as restrictive as possible. If a field doesn't absolutely need to be public, try making it private. If other classes have to get or set the field's values, provide public getter and setter methods.

DEFAULT ACCESS

A member declared without any of the words *public*, *private*, or *protected* in front of it has *default access*, also known as *package-wide access*. A member with default access is accessible only in the package in which its class resides. Figure 11-9 illustrates the point.

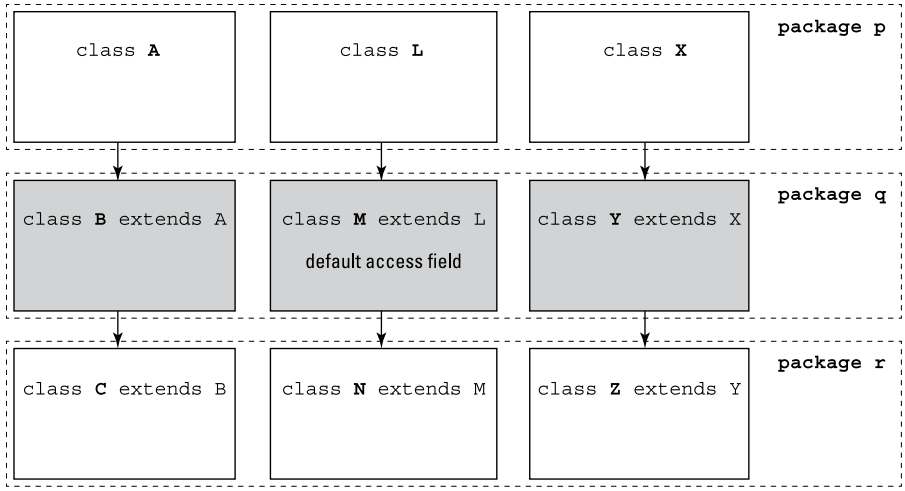


FIGURE 11-9: The range of code in which a field or method with default access can be used (shaded).

WARNING

A subpackage has no more access than any other package. Think about two packages: `com.burdbrain.drawings` and `com.burdbrain.drawings.rectangles`. One is a subpackage of the other. One lives in a `com/burdbrain/drawings` folder with the other in the `com/burdbrain/drawings/rectangles` folder. Can these packages access each other's default members? You might guess that they can. But no, they can't. A package and its subpackage have the same privileges as two unrelated packages. That's the shocking truth.

PROTECTED ACCESS

When I was first getting to know Java, I thought the word *protected* meant "nice and secure" or something like that. "Wow, that field is protected. It must be hard to get at." Well, this notion turned out to be wrong. In Java, a member that's protected is less hidden, less secure, and available for use in more classes than one that has default access. In other words, protected access is more permissive than default access. For me, the terminology is misleading. But that's the way it is.

Think of protected access this way: You start with a field that has default access (a field without the word `public`, `private`, or `protected` in its declaration). That field can be accessed only inside its own package. Now add the word `protected` to the front of the field's declaration. Suddenly, some classes outside that field's package have access to the field. You can now reference the field from a subclass of the class in which the field is declared. You can also reference the field from a sub-subclass, a sub-sub-subclass, and so on. Any descendant class will do. Figure 11-10 has the story.

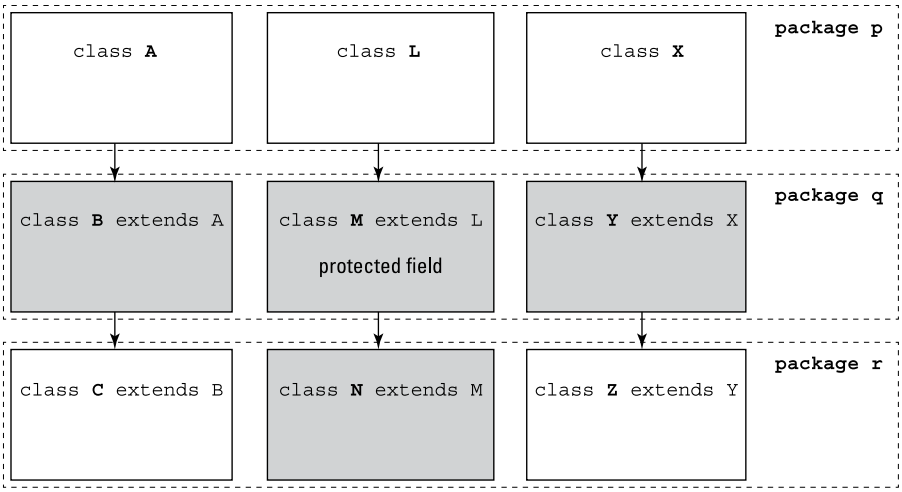


FIGURE 11-10: The range of code in which a protected field or method can be used (shaded).

Don't stress your brain trying to understand protected access. You can read about it again if you ever need to know about it.

TRY IT OUT

Here are some things for you to try:

NAME THAT TUNE

A `Work` has a title (a `String`) and a release year (an `int`).

The `Song` and `Painting` classes are subclasses of the `Work` class. Each `Song` instance has a genre (a value from an `enum` named `Genre`). The values of `Genre` are `ROCK`, `POP`, `BLUES`, and `CLASSICAL`. Each `Painting` instance has a style (a value from an `enum` named `Style`). The values of `Style` are `REALISTIC`, `ABSTRACT`, `IMPRESSIONISM`, and `EXPRESSIONISM`.

Each `Artist` instance has a name (a `String`) and a `Work` instance.

Create these classes. Devise a plausible scenario in which these classes would be spread over more than one package. In a separate class, construct instances of each class, and display information about these instances on the screen.

WHAT'S MINE IS YOURS

The following four classes live in four different `.java` files. Without typing these classes in an IDE's editor, decide which statements will cause the IDE to display error messages. For each such statement, decide on the least permissive access change that would eliminate the error message:

```
// THIS CODE DOES NOT COMPILE:

package com.mypackage.things;

import com.yourpackage.stuff.Stuff;
import com.yourpackage.stuff.morestuff.MoreStuff;

public class Things {
    protected int i = 0;
    private int j = 0;
    int k = 0;

    public static void main(String[] args) {
        var stuff = new Stuff();
        System.out.println(stuff.i);
        var moreStuff = new MoreStuff();
        System.out.println(moreStuff.i);
    }
}

package com.yourpackage.stuff;

import com.yourpackage.stuff.morestuff.MoreStuff;

public class Stuff {
    protected int i = 0;

    void aMethod() {
        new MoreStuff().myMethod();
    }
}
```

```
package com.yourpackage.stuff.morestuff;

import com.mypackage.things.Things;

public class MoreStuff extends Things {
    protected void myMethod() {
        System.out.println(i);
    }
}

package com.mypackage.things;

public class MoreThings extends Things {

    public void anotherMethod() {
        System.out.println(i);
        System.out.println(j);
        System.out.println(k);
    }
}
```

Making Static (Finding the Team Average)

Thinking about the code in Listings 11-1 and 11-2, you decide that you want to find the team's overall batting average. Rather than pore over game logs and do a careful analysis, you settle on finding the average of the numbers in Figure 11-2. That's the average of the players' averages. Using this strategy, the team in Figure 11-2 has an average of about .106. No doubt about it: The Hankies need some intensive training.

While the players are out practicing on the ball field, you have a philosophical question to answer. Where should the variables storing your overall, team-average tally go? Should you add more fields to the `Player` class in Listing 11-1? After considering this possibility, you have some reservations. An instance of the `Player` class represents just one player on the team. What business does a single player have storing overall team data? Sure, you could make the code work, but it wouldn't be an elegant solution to the problem.

Luckily for you, Java has its `static` keyword. Anything that's declared to be `static` belongs to the whole class, not to any particular instance of the class. When you create the `static` field, `totalOfAverages`, you create just one copy of the field. This copy stays with the entire class. No matter how many instances of the class you create — one, nine, or none — you have just one `totalOfAverages` field. And, while you're at it, you create another `static` field (`playerCount`) and a `static` method (`findTeamAverage`). To see what I mean, look at Figure 11-11.

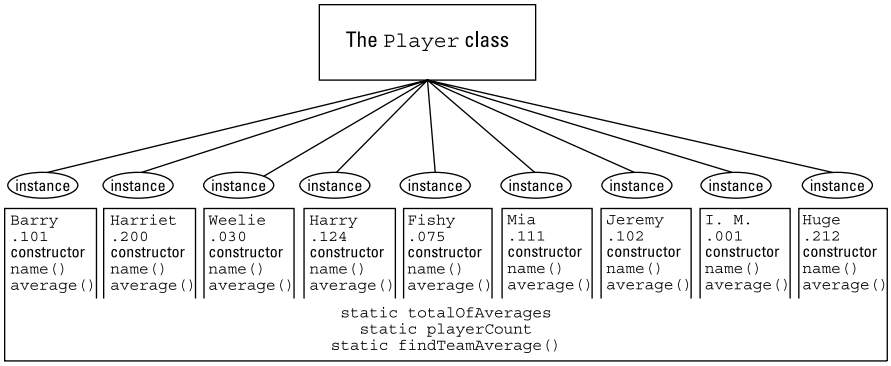


FIGURE 11-11: Some static and nonstatic fields and methods.

The `Player` class in Listing 11-1 does what it was originally supposed to do. And you remember the old saying, "If something's not broken, don't fix it." With that in mind, you create a brand-new `Player` class and put it in Listing 11-3. The new `Player` class has all the features of your old `Player` class plus a few more.

LISTING 11-3: Creating a Team Batting Average

```
package com.example.sport;

public record Player(String name, double average) {

    private static int playerCount = 0;
    private static double totalOfAverages = .000;

    public Player {
        playerCount++;
        totalOfAverages += average;
    }

    public static double findTeamAverage() {
        return totalOfAverages / playerCount;
    }
}
```

Full disclosure: In previous editions of *Java For Dummies*, the `Player` class in Listing 11-1 wasn't a record class. But in this edition, I decided to make it a record class. I did this because a 4-line record class saves space on the page. (I also wanted to show off Java's new record class feature.) Anyway, one consequence of this decision is that the code in Listing 11-3 can't extend the `Player` class in Listing 11-1. Record classes are *final*. That means you can't create a subclass of a record class. So Listing 11-3 has a completely new `Player` class. This new `Player` class has all the features of the old `Player` class (see Listing 11-1) plus the ability to calculate the overall team average. The two `Player` classes (see Listings 11-1 and 11-3) are in different packages, so Java won't mistake one of them for the other.

So much static!

In Listing 11-3, I declare `playerCount` to be `static`. That's good because something like `playerCount` (the number of team members) shouldn't belong to individual team members, and having each `Player` instance keep track of its own count would be silly. ("I know how many team members I am. I'm just one team member!") If you had nine individual `playerCount` fields, either each field would store the number 1 (which is useless) or you would have nine different copies of the count, which is wasteful and prone to error. By making `playerCount` static, you're keeping the `playerCount` in just one place, where it belongs.

The same kind of reasoning holds for the `totalOfAverages`. Eventually, the `totalOfAverages` field will store the sum of the players' batting averages. For all nine members of the Hankies, this adds up to .956.

It's not until someone calls the `findTeamAverage` method that the computer actually finds the overall Hankies team batting average. You want this `findTeamAverage` method to be static. Without the word `static`, there would be nine `findTeamAverage` methods — one for each instance of the `Player` class. This wouldn't make much sense. Each instance would have the code to calculate `totalOfAverages / playerCount` on its own, and each of the nine calculations would yield the same answer.

In general, any task that all instances have in common (and that yields the same result for each instance) should be coded as a `static` method.

Meet the compact constructor

To find the average of the team members' batting averages, you divide the total of all team members' averages by the number of team members. In Listing 11-3, that's `totalOfAverages / playerCount`. The `playerCount` and `totalOfAverages` fields in Listing 11-3 are static. Whether you have one team member, nine team members, or a thousand team members, there's only one copy of the `playerCount` field and one copy of the `totalOfAverages` field.

How do you keep tallies of `playerCount` and `totalOfAverages` values? One way is to update these fields whenever you call the `Player` constructor. If `Player` wasn't a record class, you'd code a constructor like this:

```
// A four-statement constructor

public Player(String name, double average) {
    this.name = name;
    this.average = average;
    playerCount++;
    totalOfAverages += average;
}
```

You could do this in Listing 11-3. But since Listing 11-3 is a record class, there's a better way. Remember: When you don't code a constructor in a record class, Java quietly adds its own fully parameterized implicit constructor. In other words, the code in Listing 11-3 automatically gets a constructor of the following kind:

```
public Player(String name, double average) {
    this.name = name;
    this.average = average;
}
```

To this implicit constructor, you want to add a `playerCount++` statement and a `totalOfAverages += average` statement.

Lo and behold! The wizards who design Java thought about your needs! They said you can put a *compact constructor* in your record class. In Listing 11-3, the compact constructor looks like this:

```
public Player {
    playerCount++;
    totalOfAverages += average;
}
```

A record class's compact constructor has no parentheses after the class's name. When Java sees this, it *adds* the statements in the compact constructor to the statements in its own implicit constructor. It's as if you had coded this section's 4-statement constructor. That's pretty handy.

Displaying the overall team average

Listing 11-3, shown previously, contains code that defines a `Player` class, and Listing 11-4 contains code that uses this `Player` class.

LISTING 11-4: **Using the Code from Listing 11-3**

```
import com.example.sport.Player;

import java.io.File;
import java.io.IOException;
import java.util.Scanner;

import static java.lang.System.out;

void main() throws IOException {
    var hankiesData = new Scanner(new File("Hankies.txt"));

    printTableHeader();

    for (int num = 1; num <= 9; num++) {
        var player = new Player(hankiesData.nextLine(),
                hankiesData.nextDouble());
        hankiesData.nextLine();
        printTableRow(player);
    }
    hankiesData.close();

    printTableSummary();
}

void printTableHeader() {
    out.print("Team Member");
    out.println(String.format("%40s", "Batting Average"));
    out.print("-----------");
    out.println(String.format("%40s", "---------------"));
}

void printTableRow(Player player) {
    out.print(String.format("%-40s", player.name()));
    out.println(String.format("%.3f", player.average()).substring(1));
}

void printTableSummary() {
    out.println(String.format("%44s", "----"));
    var teamAverageString = String.format("%.3f",
            Player.findTeamAverage()).substring(1);
    out.println(String.format("%44s", teamAverageString));
}
```

Figure 11-12 shows a run of the code from Listing 11-4. This run depends on the availability of the `Hankies.txt` file from Figure 11-1.

```
Team Member                          Batting Average
-------------                        -------------------
Barry Burd                                  .101
Harriet Ritter                              .200
Weelie J. Katz                              .030
Harry "The Crazyman" Spoonswagler           .124
Filicia "Fishy" Katz                        .075
Mia, Just "Mia"                             .111
Jeremy Flooflong Jones                      .102
I. M. D'Arthur                              .001
Hugh R. DaReader                            .212
                                            -----
                                            .106
```

FIGURE 11-12: A run of the code in Listing 11-4.

Listing 11-4 is rather long, but most of it is the kind of code you see in previous examples.

All about static import

Here's one new and exciting thing in Listing 11-4. It's the listing's `import static` declaration:

```
import static java.lang.System.out;
```

When I add this `import static` declaration, I can abbreviate the long-winded `System.out.print` and `System.out.println` calls.

Other import declarations begin with only the word `static`. So, why do I need the two words `import static` for a declaration in Listing 11-4?

The answer is that you can import two kinds of things — classes and static members. The `java.util` package has a *class* named `Scanner`. So, to abbreviate `java.util.Scanner` with plain old `Scanner`, you write

```
import java.util.Scanner;
```

But the `System` class (inside the `java.lang` package) has a static member named `out`:

```
public class System {
    // ...
    public static PrintStream out;
}
```

So, to abbreviate `java.lang.System.out` with plain old `out`, you write

```
import static java.lang.System.out;
```

TECHNICAL STUFF

You can always abbreviate `java.lang.System` with the simple name `System`. Java adds an implicit `import java.lang.*;` to every program.

All about static references

Notice the code

```
Player.findTeamAverage()
```

near the end of Listing 11-4.

That code has the following form:

```
ClassName.methodName()
```

In earlier chapters, I say that you normally preface a method call with an object's name, not a class's name. So why does this dot notation use a class name? The answer: When you call a `static` method, you preface the method's name with the name of the class that contains the method. The same holds true whenever you reference another class's `static` field. This makes sense because a static member belongs to an entire class, not to any particular instance of the class.

This rule about calling static methods isn't new. Chapter 7 introduces `String.format` as a way of making output look nice. When I wrote that chapter, I hoped you wouldn't notice that `String` is the name of a class, not the name of an object. But now, I can come clean. The `format` method is a static member of the `String` class.

```
public class String {
    public static String format( ... ) {
        // ...
    }
    // ...
}
```

So there! To refer to the static `format` method, you start with the name of the method's class. You write `String.format`.

Why static members don't always play well with others

In Java, every method must be a member of a class, even if that class is declared implicitly. And before Java 21, every `main` method had to be static. To honor that tradition, the `main` method in Listing 11-2 is a static member of a class named `ShowPlayers`.

```
public class ShowPlayers {

    public static void main(String[] args) throws IOException {
```

As you travel through Javaland, you'll see many static `main` methods. The thing to remember about static main methods is what other methods they can and cannot call. Take, for example, the following bad code:

```
// Kids, don't try this at home (or anywhere else).

public class Attendee {

    public static void main(String args[]) {
        var person1 = new Attendee();
        var person2 = new Attendee();
        var person3 = new Attendee();
        areYouHere();
        startMeeting();
    }

    public void areYouHere() {
        System.out.println("I'm here.");
    }

    public void startMeeting() {
        System.out.println("Welcome, everyone.");
    }
}
```

In this bad code, the main method creates three people, tries to ask whether they're attending the meeting, and then tries to start the meeting. The error message I get with this code is "Non-static method cannot be referenced from a static context." When I first started writing Java programs, I had recurring dreams about seeing this error message. So often did I see this message, so thoroughly was I perplexed, that the memory of this error message became burned into my brain.

In this `Attendee` class, the main method is static, but the `youAreHere` and `startMeeting` methods aren't static. (See Figure 11-13.) So there's only one main method, but there are three versions of the `areYouHere` method and three versions of the `startMeeting` method. When the one-and-only main method tries to call `areYouHere()`, Java doesn't know which instance's `areYouHere` method to call. The same thing happens when the static main method tries to call `startMeeting()`. In both cases, Java says "Non-static method cannot be referenced from a static context."

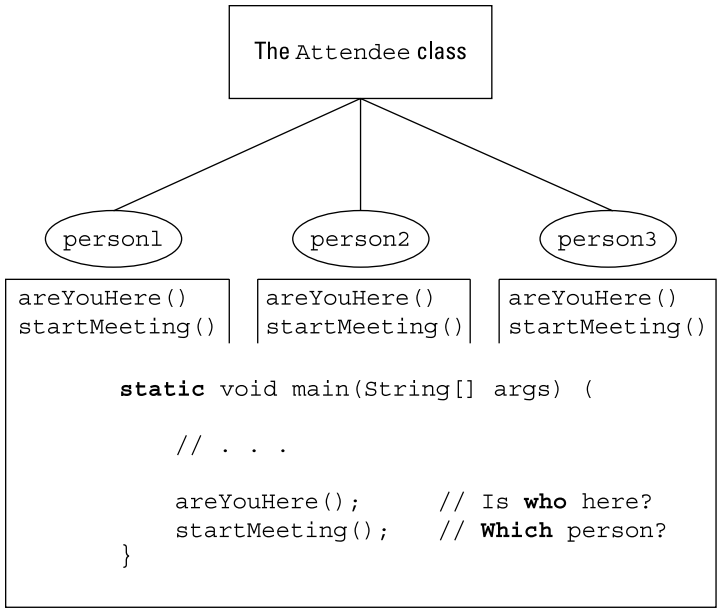


FIGURE 11-13: This is no good.

To find out who's present at a meeting, you don't stare into space and ask "Are you here?" of no one in particular. Instead, you go down your list of people on the roster and ask each of them individually "Are you here?" So each `areYouHere` call should refer to a specific attendee.

In contrast, starting the formal proceedings is a one-person job. Your code should have only one copy of the `startMeeting` method, so `startMeeting` should be static.

Here's some code with the appropriate fixes:

```
public class Attendee {

    public static void main(String args[]) {
        var person1 = new Attendee();
        var person2 = new Attendee();
        var person3 = new Attendee();
        person1.areYouHere();
        person2.areYouHere();
        person3.areYouHere();
        startMeeting();
    }

    public void areYouHere() {
        System.out.println("I'm here.");
    }

    public static void startMeeting() {
        System.out.println("Welcome, everyone.");
    }
}
```

TECHNICAL STUFF

The `startMeeting` method is a static member of the `Attendee` class. So, instead of calling `startMeeting()`, you have the option of calling `Attendee.startMeeting()`. This `Attendee` prefix is necessary when you want to call the `Attendee` class's `startMeeting` method from code that's not inside the `Attendee` class itself. (Oddly enough, a call like `person1.startMeeting()` also works, although I've never understood why that way of calling a static method makes any sense.)

TRY IT OUT

I don't know about you, but I can always use some practice with static variables and methods:

DESIGNER DUMMIES

In a previous section, you create a class to represent items in a line of men's clothing. Add fields for the name of the designer and the cost of the item. The designer's name is static because all items in the line come from the same designer — *Dummies House of Fashion*.

Write code to display eight rows. The first row displays the designer's name. The next six rows describe one complete men's wardrobe. The last row shows the wardrobe's total cost.

COUNT ME IN

In a previous section, you create a class to represent a playing card. Add a static field to your `PlayingCard` class. The field keeps track of the number of times the class's constructor has been called, and thus has a count of the number of playing cards.

ATTACK OF THE MUTANTS

What's the output of the following code? Make some predictions, and then run the code to see whether your predictions are correct:

```
package com.example.mutants;

import static java.lang.System.out;

public class Main {

    public static void main(String[] args) {

        out.println("bigValue: " + MutableInteger.bigValue);
        // out.println("bigValue: " + IntegerHolder.value); ILLEGAL

        var holder1 = new MutableInteger(42);
        var holder2 = new MutableInteger(7);

        out.println("holder1: " + holder1.value);
        out.println("holder2: " + holder2.value);
        out.println();
        holder1.value++;
        holder2.value++;
        MutableInteger.bigValue++;

        out.println("bigValue: " + MutableInteger.bigValue);
        out.println("holder1: " + holder1.value);
        out.println("holder2: " + holder2.value);

        out.println();
        holder1.bigValue++;
        out.println("bigValue according to holder1: " + holder1.bigValue);
        out.println("bigValue according to holder2: " + holder2.bigValue);
    }
}
```

```
class MutableInteger {
    int value;
    static int bigValue = 1_000_000;

    public MutableInteger(int value) {
        this.value = value;
    }
}
```

IN THIS CHAPTER

- Dealing with a bunch of objects at one time
- Creating a dictionary
- Using Java's cool functional programming features
- Developing code for multicore processors

Chapter 12
Using Collections and Streams

The English word *venery* means "the act of hunting for wild animals." In olden times, hunters made up words to refer to collections of animals. For example, a group of birds is a *flock*, and an assemblage of wolves is a *pack*. Collective nouns of this kind are called *terms of venery*. Other examples include a *pride* of lions, *gaggle* of geese, *school* of fish, *swarm* of bees, *herd* of cattle, *murder* of crows, *quiver* of cobras, *convocation* of eagles, and *litter* of cats. (Why anyone would want to go hunting for a bunch of house cats, I'll never know!)

Anyway, Java has many of its own terms of venery. But Java's terms differ from English animals' terms in an important way. Java's terms don't differ from one another by the kinds of things they describe. Instead, they differ in the way the underlying collection is organized. It's like the difference between a *committee* of people and a *mob* of people. A mob can be scary, but a committee is usually very boring.

The first part of this chapter is about Java's terms of venery, known to everyone else in the world as Java's *collection classes*. Once you have collection classes, you can take full advantage of Java's functional programming features. Those functional programming features dominate this chapter's second half.

Using an ArrayList

One of the most commonly used collection classes is the `ArrayList`. Listing 12-1 shows you how it works.

LISTING 12-1: Working with a Java Collection

```
import module java.base;

public class ShowAllWords {

    public static void main(String[] args) throws IOException {

        ArrayList<String> vocabulary = new ArrayList<>();

        var diskScanner = new Scanner(new File("AllWords.txt"));
        while (diskScanner.hasNext()) {
            vocabulary.add(diskScanner.nextLine());
        }
        diskScanner.close();

        vocabulary.remove(0);
        vocabulary.add(2, "paradox");

        for (String word : vocabulary) {
            System.out.println(word);
        }
    }
}
```

CLEARING THE CLUTTER

You may want to see a punchier, more impactful version of the code in Listing 12-1. If so, here's the simple source file:

```
void main() throws IOException {

    ArrayList<String> vocabulary = new ArrayList<>();

    var diskScanner = new Scanner(new File("AllWords.txt"));
    while (diskScanner.hasNext()) {
        vocabulary.add(diskScanner.nextLine());
    }
```

```
        diskScanner.close();

        vocabulary.remove(0);
        vocabulary.add(2, "paradox");

        for (String word : vocabulary) {
            println(word);
        }
    }
```

I'm happy to report that this shorter version does everything that the code in Listing 12-1 does. Just remember that this sidebar's code has an implicitly declared class and the ability to create such classes is a preview feature in Java 23. This feature may go away, change, or become official in later versions of Java.

For some advice about using Java's preview features, refer to Chapter 2.

The code in Listing 12-1 reads an `AllWords.txt` file and loads the file's values in an `ArrayList` named `vocabulary`. Figure 12-1 shows you a sample `AllWords.txt` file. (In case anyone cares, I got the words in this file by asking ChatGPT.) With that particular `AllWords.txt` file, the output of Listing 12-1 is what you see in Figure 12-2.

```
Luminous
harbor
zephyr
enigma
serendipity
quicksilver
echo
velvet
mirage
driftwood
```

FIGURE 12-1: Several words in a file.

According to Figure 12-1, the word at the start of the `AllWords.txt` file is `luminous`. So, you may be tempted to say the following: "Immediately after reading the contents of `AllWords.txt`, the first value in the `vocabulary` list is the word `luminous`." If you say this, you're *mostly* correct.

```
harbor
zephyr
paradox
enigma
serendipity
quicksilver
echo
velvet
mirage
driftwood
```

FIGURE 12-2: The output of the code in Listing 12-1.

Why do I say "mostly correct"? It's because the word *first* suggests "number 1," but `luminous` isn't the "number 1" value in the vocabulary list. Computer languages number everything starting with 0, not 1. Because of this, these statements are true:

- The *index* of `luminous` in the vocabulary list is 0.
- The index of the next word, `harbor`, in the vocabulary list is 1.
- The index of the next word, `zephyr`, in the vocabulary list is 2.
- And so on.

See Figure 12-3.

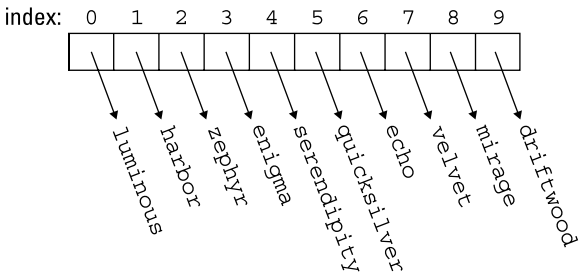


FIGURE 12-3: Immediately after reading the `AllWords.txt` file.

You can guess what happens when Java executes `vocabulary.remove(0)` in Listing 12-1. The element with index 0 (the word `luminous`) gets taken from the list. As a result, the word `harbor` moves to the 0-index slot. In fact, all remaining values move leftward to fill in the gap. After removing `luminous` from the list, you have the `vocabulary` list shown in Figure 12-4.

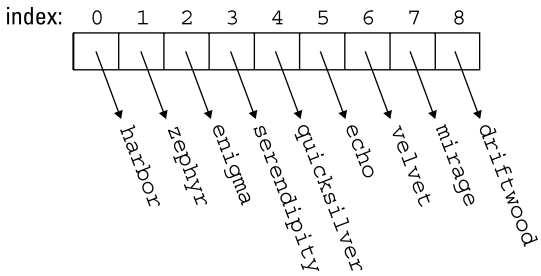


FIGURE 12-4: After removing the entry with index 0.

The next statement in Listing 12-1, `vocabulary.add(2, "paradox")`, inserts the word `paradox` into the list and gives that word the index 2. Once again, some values in the list get new indices. In particular, the words from `enigma` onward move over to accommodate the insertion. (See Figure 12-5.)

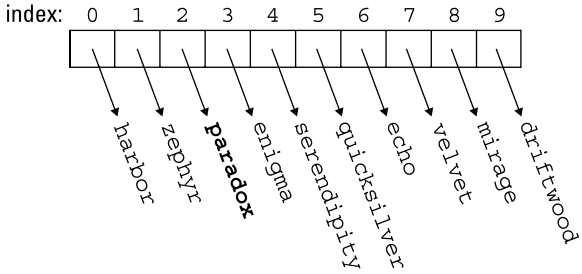


FIGURE 12-5: After inserting an entry at index 2.

Notice that an `ArrayList` instance has two different `add` methods.

- The first `add` method in Listing 12-1 has only one parameter:

```
vocabulary.add(diskScanner.nextLine());
```

  This statement takes whatever word it finds on a line of the input file and appends that word to the *end* of the list.

- The second `add` method in Listing 12-1 has two parameters:

```
vocabulary.add(2, "paradox")
```

  Rather than append a word to the end of the file, this method inserts the word at a specific index in the file.

A new way to do looping

The last few lines of Listing 12-1 contain an *enhanced* `for` *loop* — a `for` loop that doesn't use counters or indices. An enhanced `for` loop has three parts:

```
for (variableType variableName : rangeOfValues)
```

In Listing 12-1, the *variableName* is `word`, and the *variableType* is `String`. During each loop iteration, the `word` variable takes on a new `String` value. The loop's *rangeOfValues* is `vocabulary`. So, during the initial loop iteration, the value of `word` is the `vocabulary` list's initial entry, which is `"harbor"`. (Refer to Figure 12-5.) During the next iteration, the value of `word` is the list's next entry, which is `"zephyr"`. After that comes `"paradox"`. And so on.

WARNING

Java's enhanced `for` loop requires a word of caution: Each time through the loop, the variable that steps through the range of values stores a *copy* of the value in the original range. The variable does *not* point to the range itself.

For example, if you add an assignment statement that changes the value of `word` in Listing 12-1, this statement has no effect on any of the values stored in the `vocabulary` array. To drive this point home, imagine filling the `vocabulary` list with five words: `"Air"`, `"Hi"`, `"More"`, `"A"`, and `"I"`. Then you execute the following code:

```
for (var word: vocabulary) {
    word += "s";
    print(word);
    print(" ");
}

println("");

for (var word: vocabulary) {
    print(word);
    print(" ");
}
```

The `word += "s"` statement changes each `word` value *inside* the topmost `for` loop. But that statement *doesn't* change any of the `vocabulary` list's entries. So the code's output looks like this:

```
Airs His Mores As Is
Air Hi More A I
```

TIP

Java has some shortcuts for filling a new `ArrayList` with values. For example, you can write

```
ArrayList<String> list =
    new ArrayList<>(Arrays.asList("harbor", "zephyr", "enigma", "echo"));
```

Or, if you want 100 copies of the word *Dummies* in a list, you can write

```
ArrayList<String> dummiesList =
    new ArrayList<>(Collections.nCopies(100, "Dummies"));
```

Import a module in one fell swoop

The code in Listing 12-1 uses several classes from Java's API. My original draft of Listing 12-1 started with all these import declarations:

```
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
```

Eventually, I replaced these five lines with only one line: the `import module java.base` declaration at the top of Listing 12-1.

That `import module` line can replace most of the other import declarations in this book. In particular, all the declarations that begin with `import java.util` and `import java.io` can be replaced with this single `import module` declaration. The reason is that the packages whose names start with `java.util` and `java.io` are all inside the `java.base` module.

You can import the `java.base` module at the start of any file, even a file that doesn't need any imports at all. (In rare cases, you can stumble into a class name conflict. That's when you need a class in some other module that has the same name as a class in `java.base`. In that case, Java tells you that you have a problem, and you can take time to work it all out.)

The `import module` declaration is a win-win. So why don't you see it in earlier chapters? I held off using it until Chapter 12 for three reasons:

- **The import module feature is new.**

  In fact, it's a preview feature in Java 23. I don't think it will disappear in later versions of Java, but I can't be sure of it.

» **Old-fashioned import declarations (without the word `module`) occur in zillions of Java programs.**

It's good to know how those traditional import declarations work. And finally,

» **I didn't think of using import module declarations until after I wrote Chapters 1–11.**

Don't tell anybody about that.

Using generics

Look again at Listing 12-1, shown earlier, and notice the funky `ArrayList` declaration:

```
ArrayList<String> vocabulary = new ArrayList<>();
```

Each collection class is *generified.* That ugly-sounding word means that every collection declaration should contain some angle-bracketed stuff, such as `<String>`. The thing that's sandwiched between `<` and `>` tells Java what kinds of values the new collection may contain. For example, in Listing 12-1, the words `ArrayList<String> vocabulary` tell Java that `vocabulary` is a bunch of strings. That is, the `vocabulary` list contains `String` objects (not `Room` objects, not `Account` objects, not `Employee` objects — nothing other than `String` objects). So, from that point on, any reference to an item from the `vocabulary` collection is treated exclusively as a `String`. If you write

```
vocabulary.add(new Account());
```

the compiler coughs up your code and spits it out because an `Account` (created in Chapter 7) isn't the same as a `String`. (This coughing and spitting happens even if the compiler has access to the `Account` class's code — the code in Chapter 7.) But the statement

```
vocabulary.add("George Gow");
```

is just fine. Because `"George Gow"` has type `String`, the compiler smiles happily.

You have options

For historical reasons, there are a few different ways to define a new collection. For example, if you love typing, you can repeat the generic type name, like so:

```
ArrayList<String> vocabulary = new ArrayList<String>();
```

In the right situations, you can use the word `var`:

```
var vocabulary = new ArrayList<String>();
```

You may even see old code that's not generified, like this:

```
ArrayList vocabulary = new ArrayList();
```

But I don't recommend writing new declarations this way. Without generics, Java won't know for sure what type of values you're putting into your collection. So the rest of your code may require some special care. For example, you may need extra code to make sure you don't accidentally add an `Employee` instance or `Account` instance to your `vocabulary` list.

Wrap it up

In Chapter 4, I point out that Java has two kinds of types: primitive types and reference types. (If you didn't read those sections, or if you don't remember them, don't feel guilty. You'll be okay.) Things like `int`, `double`, `char`, and `boolean` are primitive types, and things like `String`, `Employee`, `ArrayList`, and `Account` are reference types.

The distinction between primitive types and reference types has been a source of contention since Java's birth in 1995. Even now, Oracle's wizards are hatching plans to work around the stickier consequences of having two kinds of types. One of those consequences is that collections, such as the `ArrayList`, can't contain values of a primitive type. For example, it's okay to write

```
ArrayList<String> vocabulary = new ArrayList<>();
```

but it's not okay to write

```
ArrayList<int> numbers = new ArrayList<>(); // BAD! BAD!
```

because `int` is a primitive type. So, if you want to store values like `3`, `55`, and `21` in an `ArrayList`, what do you do? Rather than store `int` values in the `ArrayList`, you store Java's `Integer` values:

```
ArrayList<Integer> list = new ArrayList<>();
```

In previous chapters, you see the `Integer` class in connection with the `parseInt` method:

```
int numberOfCows = Integer.parseInt("536");
```

The Integer class has many methods, such as parseInt, for dealing with int values. The class also has fields such as MAX_VALUE and MIN_VALUE, which stand for the largest and smallest values that int variables may have.

The Integer class is an example of a *wrapper class.* Each of Java's eight primitive types has a corresponding wrapper class. You can use methods and fields in Java's Double, Character, Boolean, Long, Float, Short, and Byte wrapper classes. For example, the Double class has methods named parseDouble, compareTo, toHexString, and fields named MAX_VALUE and MAX_EXPONENT.

Notice the pattern. All eight of Java's primitive types (such as int, double, and char) have names that start with lowercase letters. In contrast, all the wrapper classes (Integer, Double, Character, and so on) have names that begin with uppercase letters.

TIP

Java programmers like to begin the names of classes with uppercase letters. You can declare public class account or public class employee. But if you do, other programmers will think you're a rube.

The Integer class wraps the primitive int type with useful methods and values. In addition, you can create an Integer instance that wraps a single int value:

```
Integer myInteger = Integer.valueOf(42);
```

In this line of code, the myInteger variable has one int value inside it: the int value 42. In Paul's words, wrapping the int value 42 into an Integer object myInteger is "something like putting lots of extra breading on okra — it makes 42 more digestible for finicky eaters like collections."

Instances of the other wrapper classes work the same way. For example, an instance of the Double class wraps up a single primitive double value:

```
Double averageNumberOfTomatoes = Double.valueOf(1.41421356237);
```

Here's a program that stores five Integer values in an ArrayList:

```
void main() {
    ArrayList<Integer> list = new ArrayList<>();
    fillTheList(list);
    for (Integer n : list) {
        println(n);
    }
}
```

```
public static void fillTheList(ArrayList<Integer> list) {
    list.add(85);
    list.add(19);
    list.add(0);
    list.add(103);
    list.add(13);
}
```

In the code, notice calls like `list.add(85)` that have `int` value parameters. At this point, little Billy gets excited and says, "Look, Mom! I added the primitive `int` value `85` to my `ArrayList`!" No, Billy. That's not what's going on.

In this code, the `list` collection contains `Integer` values, not `int` values. A primitive `int` value is a lot like an instance of the `Integer` class. But a primitive `int` value isn't exactly the same as an `Integer` instance.

What's going on is called *autoboxing*. Before Java 5.0, you had to write

```
list.add(new Integer(85));
```

if you wanted to add an `Integer` to an `ArrayList`. But Java 5.0 and later Java versions can automatically wrap an `int` value inside a box. An `int` value in a parameter list becomes an `Integer` in `ArrayList`. Java's autoboxing feature makes programs easier to read and write.

Are we done yet?

Here's a pleasant surprise. When you write a program like the one shown previously, in Listing 12-1, you don't have to know how many names are in the input file. Having to know the number of names may defeat the purpose of using the easily expandable `ArrayList` class. Rather than loop until you read exactly nine names, you can loop until you run out of data.

The `Scanner` class has several nice methods, such as `hasNextInt`, `hasNextDouble`, and plain old `hasNext`. Each of these methods checks for more input data. If there's more data, the method returns `true`. Otherwise, the method returns `false`.

Listing 12-1 uses the general-purpose `hasNext` method. This `hasNext` method returns `true` as long as there's anything more to read from the program's input. After the program scoops up that last `driftwood` line in Figure 12-1, the subsequent `hasNext` call returns `false`. This `false` condition ends execution of the `while` loop and plummets the computer toward the remainder of the Listing 12-1 code.

The `hasNext` method is quite handy. In fact, `hasNext` is so handy that it's part of a bigger concept known as an *iterator*, and iterators are baked into all of Java's collection classes.

Once and again

An iterator spits out a collection's values, one after another. To obtain a value from the collection, you call the iterator's `next` method. To find out whether the collection has any more values in it, you call the iterator's `hasNext` method. For example, you can replace the loop at the end of Listing 12-1 with the following code:

```
Iterator<String> iterator = vocabulary.iterator();
while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
```

The first line of code creates an iterator from the `vocabulary` collection. The second and third lines call the iterator's `hasNext` and `next` methods to grab all objects stored in the `vocabulary` collection — one for each iteration of the loop. These lines display each of the `vocabulary` collection's values. When you make this code replacement, you get the same output as before. (You get the output in Figure 12-2.)

REMEMBER

To run this code, you may need an `import java.util.Iterator` declaration.

Which is better? An enhanced `for` loop or an iterator? Java programmers prefer the enhanced `for` loop because the `for` loop involves less baggage — no `iterator` object to carry from one line of code to the next. But as you can see later in this chapter, the most programming-enhanced feature can be upgraded, streamlined, tweaked, and otherwise reconstituted. There's no end to the way you can improve upon your code.

Sorting and Searching

Every introductory computer programming book has sections featuring two important problems:

- **Sorting:** Reorder the values in a list from increasing to decreasing, or from decreasing to increasing.
- **Searching:** Find a particular value that's one of many values in a list. Report the location of that value in the list.

Java has methods to solve both of these problems easily. You can see them in Listing 12-2.

LISTING 12-2: Where in the World Is the Word "Enigma"?

```
void main() throws IOException {
    ArrayList<String> vocabulary = new ArrayList<>();

    var diskScanner = new Scanner(new File("AllWords.txt"));
    while (diskScanner.hasNext()) {
        vocabulary.add(diskScanner.nextLine());
    }
    diskScanner.close();

    Collections.sort(vocabulary);
    for (String word : vocabulary) {
        println(word);
    }

    println("");
    print("Enigma's index: ");
    println(vocabulary.indexOf("enigma"));
}
```

Figure 12-6 shows a run of the code in Listing 12-2.

```
driftwood
echo
enigma
harbor
luminous
mirage
quicksilver
serendipity
velvet
zephyr

Enigma's index: 2
```

FIGURE 12-6: Sorted and searched.

In Listing 12-2, the call `Collections.sort(vocabulary)` rearranges the values in the `vocabulary` list. This call uses the default arrangement, which is alphabetical ordering.

Later in Listing 12-2, the call `vocabulary.indexOf("enigma")` returns the index of the first occurrence of the word `"enigma"` in the sorted `vocabulary` list. In Figure 12-6, `driftwood` has index 0, `echo` has index 1, and `enigma` has index 2. And, sure enough, a call to `indexOf("enigma")` returns the `2` that you see at the end of Figure 12-6.

TIP

Java's sorting methods are versatile. You can arrange values in any order you want. For example, the call `Collections.reverse(vocabulary)` arranges an alphabetically sorted list in reverse alphabetical order. And the call

```
Collections.sort(vocabulary,
        Comparator.comparingInt(w -> w.charAt(w.length() - 1)));
```

arranges values based on the *last* letter in each word.

TECHNICAL STUFF

Chapter 11 describes Java's `static` keyword and reveals some of the things you do when you bump into a static field or method. In Listing 12-1, you write `Collections.sort` because `sort` is a `static` method in Java's `Collections` class. My college students freeze up when I use the word `static`. So, if you don't understand this paragraph, don't worry about it. You can learn about `static` as you write more Java programs.

Another Kind of Collection Class

I tried to find out how many collection classes the Java 23 API has, but I gave up after a while. It has too many for a simple tally. Anyway, I decided to show off a class whose structure differs significantly from the `ArrayList` class. It's Java's `HashMap` class.

A `HashMap` instance stores a lookup table. For example, Dr. Rachael Tatman counted the number of occurrences of 333,333 words on Internet pages and published her results at `www.kaggle.com/datasets/rtatman/english-word-frequency`. I copied ten of her values to create a file named `WordFrequencies.txt`. (In retrospect, I should have asked her to count the number of collection classes in Java's API!) My `WordFrequencies.txt` file is shown in Figure 12-7.

The code in Listing 12-3 reads the `WordFrequencies.txt` file into a hash map. Then the program tries to find two words in the hash map and output those numbers' frequencies.

```
luminous      1521841
harbor       14117675
zephyr        1334891
enigma        1597665
serendipity    830226
quicksilver    808862
echo         19410462
velvet        5591498
mirage        2493561
driftwood      536080
```

FIGURE 12-7: The `Word Frequencies .txt` file.

LISTING 12-3: **Using a HashMap**

```
void main() throws IOException {
    var diskScanner = new Scanner(new File("WordFrequencies.txt"));

    var map = new HashMap<String, Integer>();

    while (diskScanner.hasNext()) {
        map.put(diskScanner.next(), diskScanner.nextInt());
    }
    diskScanner.close();

    println(map.get("quicksilver"));
    println(map.get("dummies"));
}
```

For a run of the code in Listing 12-3, see Figure 12-8.

```
808862
null
```

FIGURE 12-8: Running the code in Listing 12-3.

Listing 12-3 creates a variable whose type is `HashMap<String, Integer>`. So the map is shaped like a 2-column table, with `String` values in the first column and `Integer` values in the second column. It's the same shape as the text in Figure 12-7. (If you're not sure why the type declaration says `<String, Integer>` instead of `<String, int>`, refer to this chapter's earlier section "Wrap it up.")

The most important methods in the `HashMap` class are `put` and `get`. You call `put` to add a row to the map and `get` to fetch a value from the map.

For example, the loop in Listing 12-3 fills the map by calling `put` several times. Each time, the call's parameters are a `String` value (like the word `"luminous"`),

and an `Integer` value (like the number 1521841). When the loop is finished reading the file in Figure 12-7, the `map` variable's `HashMap` instance has ten word/number pairs.

To find out how many times Dr. Tatman found the word *quicksilver*, you call `map.get("quicksilver")` near the end of Listing 12-3. According to Figure 12-8, she found *quicksilver* 808,862 times.

The last statement in Listing 12-3 looks for the frequency of the word *dummies*. In fact, Dr. Tatman's research saw the word *dummies* 3,392,894 times, but I didn't put that information in my `WordFrequencies.txt` file. So Listing 12-3 doesn't add `"dummies"` to the program's hash map. As a result, the call to `map.get("dummies")` returns nothing at all. And, in Java, the word `null` means "nothing at all."

So many collection classes!

The `ArrayList` class that I use in many of this chapter's examples is only the tip of the Java collections iceberg. The Java library contains many collections classes, each with its own advantages. Table 12-1 contains an abbreviated list.

TABLE 12-1: Some Collection Classes

Class Name	Characteristic
`ArrayList`	A resizable list.
`LinkedList`	A list of values, each having a field that points to the next one in the list.
`Stack`	A structure that grows from bottom to top. The structure is optimized for access to the topmost value. You can easily push a value onto the top or pop off the topmost value.
`Queue`	A structure that grows at one end. The structure is optimized for adding values to one end (the rear) and removing values from the other end (the front).
`PriorityQueue`	A structure, like a queue, that lets certain (higher-priority) values move toward the front.
`HashSet`	A collection containing no duplicate values in no particular order.
`HashMap`	A collection of key/value pairs.

To find out which collection classes best meet your needs, visit `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Collection.html`. (Warning: This page is difficult to read!)

TRY IT OUT

Once again, I'd like to put you to work:

OUR BEST PLAYERS

Use the `Player` class from Chapter 11 to make an `ArrayList` of `Player` objects. Write a program to display the names of the players whose average is .100 or higher.

OUR BIGGEST NUMBER

Create an `ArrayList` containing `Integer` values. Then step through the values in the list to find the largest value among all values in the list. For example, if the list contains the numbers 85, 19, 0, 103, and 13, display the number 103.

INSERT IN ORDER

Create an `ArrayList` containing `String` values in alphabetical order. When the user types an additional word on the keyboard, the program inserts the new word into the `ArrayList` in the proper (alphabetically ordered) place.

For example, imagine that the list starts off containing the words `"cat"`, `"dog"`, `"horse"`, and `"zebra"` (in that order). After the user types the word `fish` on the keyboard (and presses Enter), the list contains the words `"cat"`, `"dog"`, `"fish"`, `"horse"`, and `"zebra"` (in that order).

To write this program, you may find the `String` class's `compareToIgnoreCase` method and the `ArrayList` class's `size` method useful. You can find out about these methods by visiting `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html#compareToIgnoreCase(java.lang.String)` and `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/ArrayList.html#size()`.

A REAL DICTIONARY

Modify the code in Listing 12-3 so that the map contains ten words and their definitions. When the user types one of the ten words, the program displays that word's definition.

Functional Programming

From 1953 to 1957, John Backus and others developed the FORTRAN programming language, which contained the basic framework for thousands of today's programming languages. The framework has come to be known as *imperative programming* because of its do-this-then-do-that nature.

A few years after the rise of FORTRAN, John McCarthy created another language, named *Lisp*. Unlike FORTRAN, the underlying framework for Lisp is *functional programming.* In a purely functional program, you avoid writing "do this, then do that." Instead, you write things like "Here's how you'll be transforming this into that when you get around to doing the transformation."

For one reason or another, imperative programming became the dominant mode. But in recent years, functional programming has emerged as a powerful and useful way of thinking about code.

Problem-solving the old-fashioned way

Congratulations! Your license to sell goods over the Internet has been approved. You've set up an e-commerce site and paid for some advertising. Your inventory includes joke books and novelty prank toys. One of the toys is a can with a label that says Cashew Nuts. When someone opens it, a fake, spring-loaded snake pops out. Another of your toys is a small buzzer. When you shake someone's hand, you surprise them with a fast vibration. You also sell traditional Groucho eyeglasses. Whoever wears them appears to have bushy eyebrows and a big mustache.

In your world, the sale of a single item looks something like the stuff in Listing 12-4. Each sale has an item and a price.

LISTING 12-4: The Sale Class

```
package com.example.sales;

public record Sale(String item, double price) {
}
```

To make use of the `Sale` class, you create a small program. The program totals up the sales on books. The program is shown in Listing 12-5.

LISTING 12-5: Total Sales of Books

```
import com.example.sales.Sale;

void main() {
    ArrayList<Sale> sales = new ArrayList<>();
    var currency = NumberFormat.getCurrencyInstance();

    fillTheList(sales);
```

```
    double runningTotal = 0;
    for (Sale sale : sales) {
        if (itemIsBook(sale)) {
            runningTotal += sale.price();
        }
    }
    println(currency.format(runningTotal));
}

boolean itemIsBook(Sale sale) {
    return sale.item().equals("Book");
}

void fillTheList(ArrayList<Sale> sales) {
    sales.add(new Sale("Book", 10.95));
    sales.add(new Sale("Book", 30.00));
    sales.add(new Sale("Toy", 9.67));
    sales.add(new Sale("Book", 20.00));
}
```

The output of the program in Listing 12-5 is `$60.95`. That's the total revenue from all books sold. On my computer, the combination of

```
var currency = NumberFormat.getCurrencyInstance();
```

and

```
currency.format(runningTotal)
```

makes that total look like a US dollar amount.

You can customize the display for your own country's currency. For example, with the code

```
var currency = NumberFormat.getCurrencyInstance(Locale.JAPAN);
```

the program's output is `¥61`.

A familiar problem

The scenario in Listing 12-5 isn't unusual. You have a collection of items (a collection of sales, perhaps). You step through the items in the collection, finding the items that meet certain criteria (the sale of a book, for example). You grab a certain value (such as the sale price) of each item that meets your criteria. Then you

do something useful with the values you've grabbed (for example, adding the values together).

Here are some other examples:

- **Step through your list of employees.** Find each employee whose performance evaluation scored 3 or higher. Give each such employee a $100 bonus and then determine the total amount of money you'll pay in bonuses.
- **Step through your list of customers.** For each customer who has shown interest in buying a smartphone, send the customer an email about this month's discount plans.
- **Step through the list of planets that have been discovered.** For each M-class planet, find the probability of finding intelligent life on that planet. Then find the average of all such probabilities.

This scenario is so common that it's worth finding better and better ways to deal with the scenario. One way to deal with it is to use some of the functional programming features in Java.

Lambda expressions

In Chapter 3 of *Java For Dummies*, the author says,

> *In Java, a method is a list of things to do. Every method has a name, and you tell the computer to do the things in the list by using the method's name in your program.*

He's a clever fellow because, in this sentence, he's careful to use the word *method.* Yes, a method is a list of things to do, but not all to-do lists are methods, and not all to-do lists have names.

A *lambda expression* is a to-do list with no name. Here's an example:

```
sale -> sale.item().equals("Book")
```

When you put this expression in just the right place, the expression does the work of the `itemIsBook` method in Listing 12-5. Figure 12-9 brings this point home.

A lambda expression is a concise way of creating a list of things to do. The list may include only one to-do item (as in Figure 12-9) or any number of items:

```
sale -> {
    var compareTo = "Book";
```

```
    return sale.item().equals(compareTo);
}
```

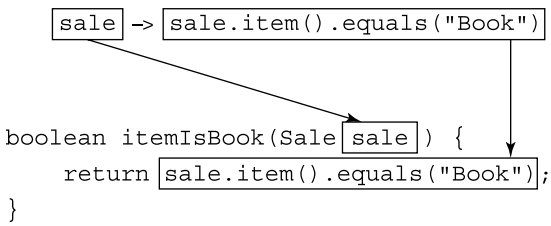


FIGURE 12-9: Does the item that's being sold happen to be a book?

When you put more than one to-do item in a lambda expression, the expression must have a complete method body with curly braces, semicolons, and the `return` keyword. Only the stuff before the arrow (`->`) is abbreviated.

The lambda expression in Figure 12-9 has only one parameter: the `sale` parameter. Can a lambda expression have more than one parameter? Of course, it can. Here's an example:

```
(price1, price2) -> price1 + price2
```

When you have more than one parameter, you have to surround the parameters with parentheses and separate them with commas.

Figure 12-10 describes the new lambda expression's meaning.

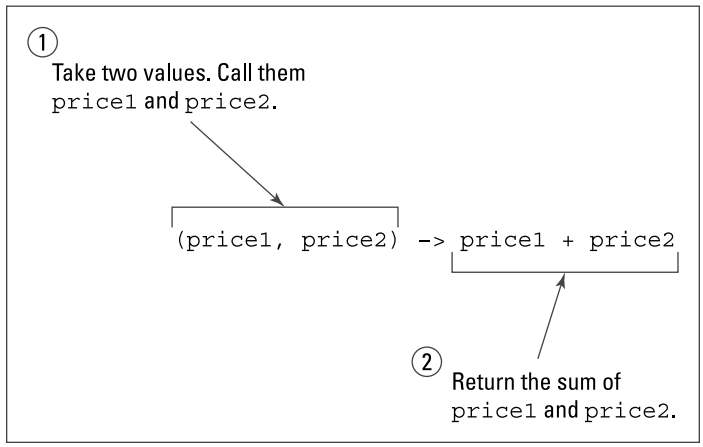


FIGURE 12-10: Add two prices.

The lambda expression in Figure 12-10 does (roughly) what the following method does:

```
double sum(double price1, double price2) {
    return price1 + price2;
}
```

The interpretation of streams

The earlier section "Once and again" introduces iterators. An iterator's `next` method spits out a collection's values, one by one. That's helpful, but Java takes this concept a step further with the notion of a stream. A *stream* is like an iterator except that, with a stream, you don't have to call a `next` method. A stream spits out a collection's values automatically. To get values from a stream, you don't call a stream's `next` method. Instead, you create a pipeline that pulls values from the stream. This pipeline solution uses concepts from functional programming.

A pipeline consists of several methods. Each method takes the data, transforms the data in some way or another, and hands its results to the next method in line. Figure 12-11 illustrates the pipeline for this chapter's sales problem.

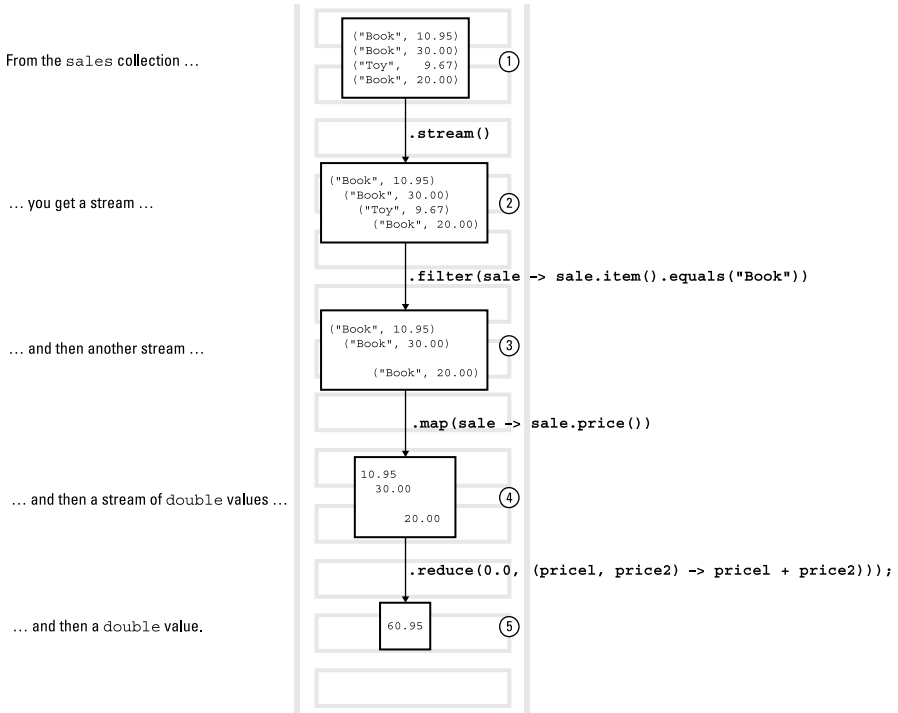


FIGURE 12-11: A functional programming pipeline.

In Figure 12-11, imagine a bunch of raw materials as they're transformed along an assembly line. Each downward-pointing arrow in the figure represents a method (or, metaphorically, a worker on the assembly line).

For example, in the transition from the figure's second box to the third box, a worker method (the `filter` method) sifts out sales of items that aren't books. Imagine Lucy Ricardo (in the classic "I Love Lucy" episode at the chocolate factory) standing between the second and third boxes, removing each toy from the assembly line and tossing it carelessly onto the floor.

Taken as a whole, the entire pipeline shown in Figure 12-11 adds up the prices of books sold. Listing 12-6 contains a complete program using the streams and lambda expressions of Figure 12-11.

LISTING 12-6: **Living the Functional Way of Life**

```
import com.example.sales.Sale;

void main() {
    ArrayList<Sale> sales = new ArrayList<>();
    var currency = NumberFormat.getCurrencyInstance();

    fillTheList(sales);

    double total = sales.stream()
            .filter(sale -> sale.item().equals("Book"))
            .map(sale -> sale.price())
            .reduce(0.0, (price1, price2) -> price1 + price2);

    println(currency.format(total));
}

void fillTheList(ArrayList<Sale> sales) {
    sales.add(new Sale("Book", 10.95));
    sales.add(new Sale("Book", 30.00));
    sales.add(new Sale("Toy", 9.67));
    sales.add(new Sale("Book", 20.00));
}
```

The output of the program in Listing 12-6 is `$60.95` (the same as the output of Listing 12-5).

The boldface code in Listing 12-6 is one big Java assignment statement. The stuff after the equal sign is a sequence of method calls. Each method call passes items to the next method call in line. That's how you form the pipeline.

The next few sections describe each of the method calls.

Getting a stream from a collection

When you do functional programming in Java, you start with a collection and call the `stream` method on that collection. This gives you a stream that feeds items from the collection to the rest of the code's pipeline. For example, in Listing 12-6, the name `sales` refers to an `ArrayList`, and `sales.stream()` refers to a stream. (See Figure 12-12.)

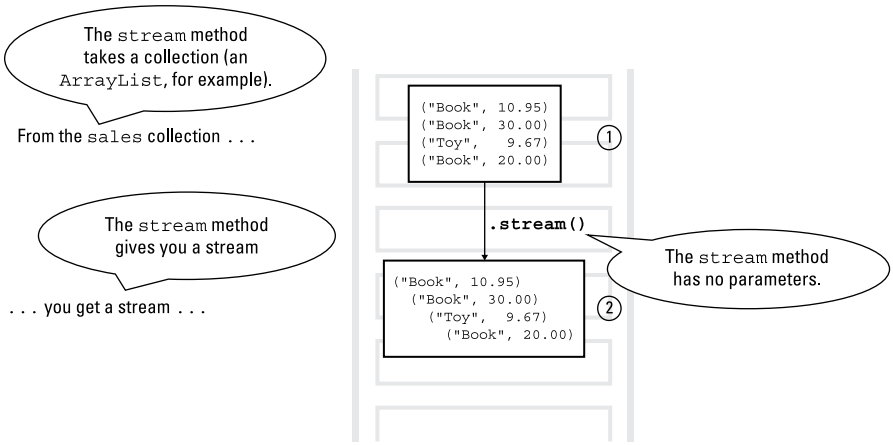


FIGURE 12-12: Yet another streaming service.

Filtering the items in a stream

A `filter` method takes items from one stream and passes some of them to another stream. The `filter` method's parameter is a condition, and the items that get passed are the ones that satisfy the condition. For example, in Listing 12-6, the lambda expression

```
sale -> sale.item().equals("Book")
```

returns `true` for a particular sale item as long as that sale happens to be a book.

Figure 12-13 illustrates the work of the `filter` method in Listing 12-6.

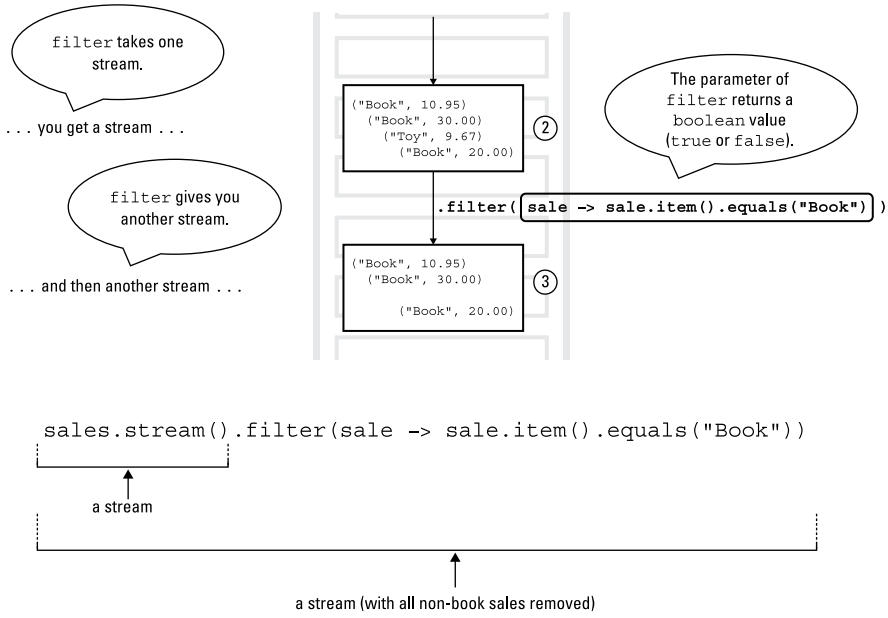

filter takes one stream.
. . . you get a stream . . .
filter gives you another stream.
. . . and then another stream . . .
("Book", 10.95)
("Book", 30.00)
("Toy", 9.67)
("Book", 20.00)
2
The parameter of filter returns a boolean value (true or false).
.filter(sale -> sale.item().equals("Book"))
("Book", 10.95)
("Book", 30.00)
("Book", 20.00)
3
sales.stream().filter(sale -> sale.item().equals("Book"))
a stream
a stream (with all non-book sales removed)

FIGURE 12-13: We don't care about the toys.

Getting a value from each item in a stream

A `map` method takes each item in a stream, does something with that item, and then passes the "something" to the next stream in the pipeline. For example, with each sale, the `map` method in Listing 12-6 gets the price of the sale and passes that price onward.

Figure 12-14 illustrates the work of the `map` method in Listing 12-6.

Summarizing the items in a stream

A `reduce` method combines a stream's items. The reduce call in Listing 12-6 adds all the values from its incoming stream — the stream that comes from the `map` method. That incoming stream has the prices of all the books from the original `sales` list. (See Figure 12-15.)

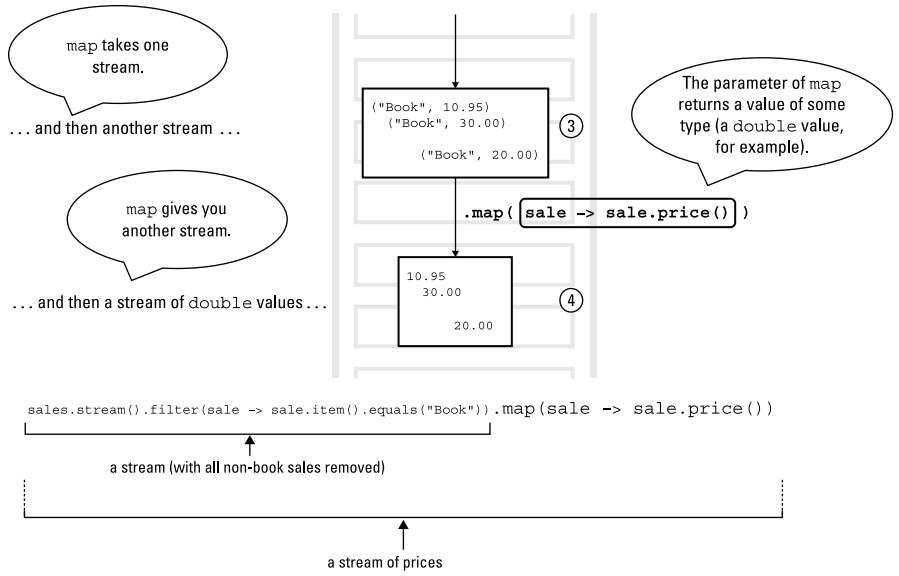

map takes one stream.
. . . and then another stream . . .
map gives you another stream.
. . . and then a stream of double values . . .
("Book", 10.95)
("Book", 30.00)
("Book", 20.00)
3
The parameter of map returns a value of some type (a double value, for example).
.map(sale -> sale.price())
10.95
30.00
20.00
4
sales.stream().filter(sale -> sale.item().equals("Book")).map(sale -> sale.price())
a stream (with all non-book sales removed)
a stream of prices

FIGURE 12-14: Getting the price from each book sale.

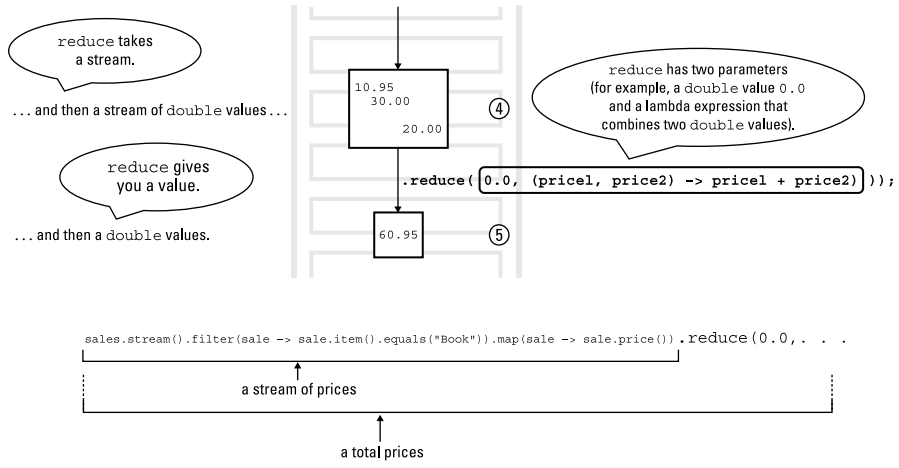

reduce takes a stream.
. . . and then a stream of double values . . .
reduce gives you a value.
. . . and then a double values.
10.95
30.00
20.00
4
reduce has two parameters (for example, a double value 0.0 and a lambda expression that combines two double values).
.reduce(0.0, (price1, price2) -> price1 + price2));
60.95
5
sales.stream().filter(sale -> sale.item().equals("Book")).map(sale -> sale.price()).reduce(0.0, . . .
a stream of prices
a total prices

FIGURE 12-15: Getting the total price of all book sales.

Java's `reduce` method does more than just scrunch numbers together. The method takes two parameters:

- The first parameter is an initial value.

  In Figure 12-15, the initial value is `0.0`.

- The second parameter is a function (usually, a lambda expression) that takes two of its own parameters and returns yet another value. For example, in Figure 12-15, the lambda expression takes two parameters, `price1` and `price2`, and returns the sum:

```
(price1, price2) -> price1 + price2
```

The lambda expression combines the values from the incoming stream. The initial value serves as the starting point for all the combining. So, in Figure 12-15, the `reduce` method performs three additions. (See Figure 12-16.)

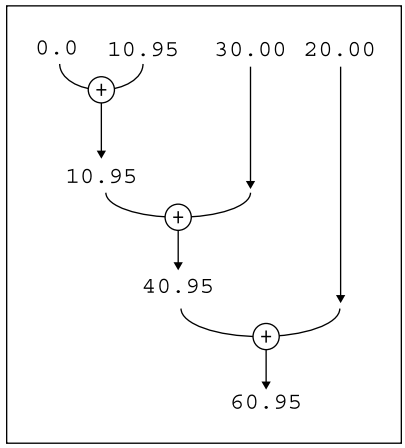


FIGURE 12-16: The reduce method adds two values at a time.

For comparison, imagine calling the method

```
reduce(10.0, (value1, value2) -> value1 * value2)
```

with the stream whose values include `3.0`, `2.0`, and `5.0`. The resulting action is shown in Figure 12-17.

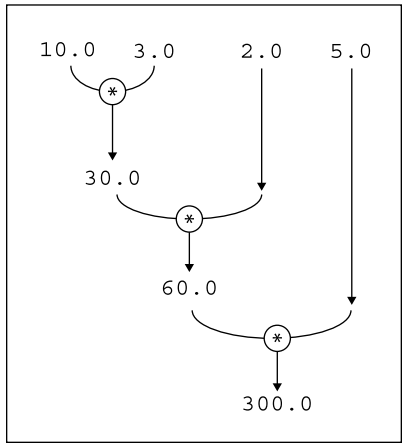


FIGURE 12-17: The reduce method multiplies values from an incoming stream.

The `reduce` method plays a special role in Listing 12-6. Every method in a stream performs a kind of operation: either an intermediate operation or a terminal operation. An *intermediate* operation is one that returns another stream. A *terminal* operation is one that either returns something other than a stream or returns nothing (`void`).

A chain of calls like the one in Listing 12-6 begins with one or more intermediate operations and ends with a single terminal operation. For example, in Listing 12-6, `filter` and `map` perform intermediate operations, and `reduce` performs a terminal operation. In a sense, the terminal operation pulls values from all the intermediate operations. When Java tries to find the value of `total`, the terminal `reduce` operation eagerly asks for `price1` and `price2`. At that moment, the lazy, intermediate `filter` and `map` operations wake up and start pumping out values.

REMEMBER

If you don't put a terminal operation call at the end of your call chain, Java doesn't execute any of the intermediate calls. For an example, see the Try It Out paragraph at the end of this chapter.

Why bother?

The chain of method calls in Listing 12-6 accomplishes everything that the loop in Listing 12-5 accomplishes. But the code in Figure 12-14 uses concepts from functional programming. What's the big deal? Are you better off with Listing 12-6 than with Listing 12-5?

You are. For the past several years, the big trend in chip design has been *multicore* processors. With several cores, a processor can execute several statements at the same time, speeding up a program's execution by a factor of 2 or 4 or 8 or even more. Programs run much faster if you divide the work among several cores. But how do you divide the work?

You can modify the imperative code in Listing 12-5. For example, with some fancy features, you can hand different loop iterations to different cores. But the resulting code is messy. For the code to work properly, you have to micromanage the loop iterations, checking carefully to make sure that the final `runningTotal` value is correct.

In contrast, the functional code is easy to modify. To take advantage of multicore processors, you change *only one word* in Listing 12-6!

```
double total = sales.parallelStream()
        .filter(sale -> sale.item().equals("Book"))
        .map(sale -> sale.price())
        .reduce(0.0, (price1, price2) -> price1 + price2);
```

In Listing 12-6, the `stream()` method call creates a serial stream. With a *serial* stream, Java does its processing one sale at a time. But a call to `parallelStream()` creates a slightly different kind of stream: a parallel stream. With a *parallel* stream, Java divides the work among the number of cores in the computer's processor (or according to some other useful measure of computing power). If you have 4 million sales and four cores, each core processes 1 million of the sales.

Each core works independently of the others, and each core dumps its result into a final `reduce` method. The `reduce` method combines the cores' results into a final tally. In the best possible scenario, all the work gets done in one-fourth the time it would take with an ordinary serial stream.

TECHNICAL STUFF

When you read the preceding paragraph, don't gloss over the phrase *best possible scenario.* Parallelism isn't magic. And sometimes, parallelism isn't your friend. Consider the situation in which you have only 20 sale amounts to tally. The time it takes to divide the problem into four groups of 5 sales each far exceeds the amount of time you save in using all four cores. In addition, some problems don't lend themselves to parallel processing. Imagine that the price of an item depends on the number of similar items being sold. In that case, you can't divide the problem among four independently operating cores. If you try, each core has to know what the other cores are doing. You lose the advantage of having four threads of execution.

NO VARIABLES? NO PROBLEM!

In imperative programming, your code's pieces interact with one another. All the pieces might be updating the current price of Oracle stock shares (ticker symbol: ORCL). The simultaneous updates may conflict with one another and result in an incorrect outcome. You've experienced the same phenomenon if you've ever clicked a website's Purchase button, only to learn that the item you're trying to purchase is out of stock. Someone else completed a purchase while you were filling in your credit card information. Too many customers were grabbing for the same goods at the same time.

The source of the problem is *shared data.* How many clients share simultaneous access to Oracle's stock price? How many customers share access to a web page's Purchase button? Today's multicore processors can perform more than one instruction at a time. How many simultaneous instructions can all be trying to modify the same variable's value?

In imperative programming, a variable is a place where statements share their values with one another. Can you avoid using variables in your code?

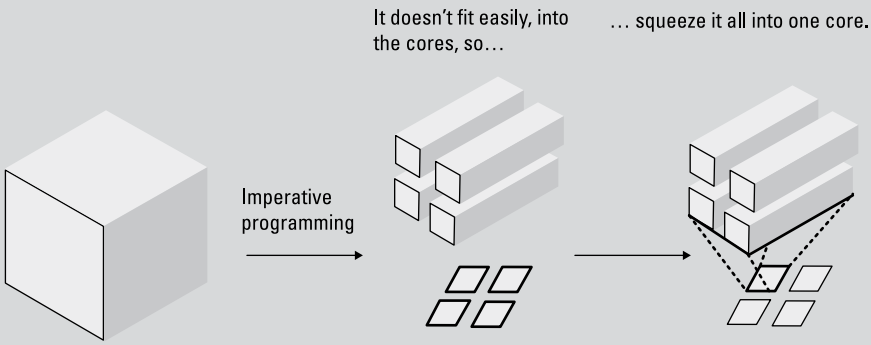
(continued)

(continued)

Compare the loop in Listing 12-5 with the functional programming code in Listing 12-6. In Listing 12-5, the `runningTotal` variable is shared among all loop iterations. Because each iteration can potentially change the value of `runningTotal`, you can't assign each iteration to a different processor core. If you did, you'd risk having two cores updating that running total at the same time. (Chances are good that, because of the simultaneous updating, neither core would do its update correctly!) But the `filter`/`map`/`reduce` lines in Listing 12-6 make no reference to the code's `total` variable. The program's `total` variable is a final tally, not a running total. In fact, none of the variables in Listing 12-6 represents a running total. In Listing 12-6, the running total is completely anonymous.

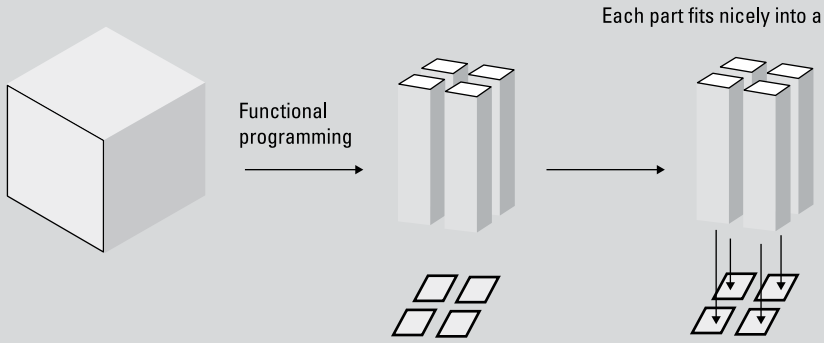
In imperative programming, a variable is a place where statements share their values with one another. But functional programming shuns variables. So, when you do functional programming, you don't have a lot of data sharing. Many of the difficulties associated with shared data vanish into thin air.

Here's an analogy: Imagine a programming problem as a cube, and then imagine an imperative programming solution as a way of slicing the cube into manageable pieces. (See the first sidebar figure.)



To get the most out of a four-core processor, you divide your code into four pieces — one piece for each core. But with imperative programming, your program's pieces don't fit neatly into your processor's cores. The figure shows what happens when you try to squeeze an imperative program into a multicore processor.

Functional programming also divides code into pieces, but it does so along different lines. With functional programming, the pieces of the code fit neatly into the processor's cores. (See the second sidebar figure.)



Functional code eliminates much of the worry about sharing data among the parts of a program. In that way, functional programming is an elegant solution to an important computing problem.

Method references

Take a critical look at the last lambda expression in Listing 12-6:

```
(price1, price2) -> price1 + price2)
```

This expression does roughly the same work as a `sum` method. (In fact, you can find a `sum` method's declaration in the earlier section "Lambda expressions.") If your choice is between typing a 3-line `sum` method and typing a 1-line lambda expression, you'll probably choose the lambda expression. But what if you have a third alternative? Rather than type your own `sum` method, you can refer to an existing `sum` method. Using an existing method is the quickest and safest thing to do.

As luck would have it, Java's `Double` class contains a static `sum` method. You don't have to create your own `sum` method. If you run the following code:

```
double i = 5.0, j = 7.0;
println(Double.sum(i, j));
```

the computer displays `12.0`. So, rather than type the `price1 + price2` lambda expression in Listing 12-6, you can use two colons to create a *method reference* — an expression that refers to an existing method.

```
sales.stream()
    .filter(sale -> sale.item().equals("Book"))
```

```
    .map(sale -> sale.price())
    .reduce(0.0, Double :: sum);
```

The expression `Double :: sum` refers to the `sum` method belonging to Java's `Double` class. When you use this `Double :: sum` method reference, you do the same thing that the last lambda expression does in Listing 12-6. Everybody is happy.

For information about static methods, see Chapter 11.

Some black sheep among the lambdas

In this chapter, I've saved the worst for last. I refuse to give the following code a listing number. Instead, I refer to it as the "Ugly Code."

UGLY CODE:

Lambda Expressions with Side Effects

```
// This code works correctly, but its style is terrible.

import com.example.sales.Sale;

double runningTotal = 0.0;

void main() {
    ArrayList<Sale> sales = new ArrayList<>();
    var currency = NumberFormat.getCurrencyInstance();

    fillTheList(sales);

    var kindOfItem = "Book";

    sales.stream()
            .filter(sale -> sale.item().equals(kindOfItem))
            .map(sale -> sale.price())
            .forEach(price -> addToRunningTotal(price));


    println(currency.format(runningTotal));
}

void addToRunningTotal(double price) {
    runningTotal += price;
}

void fillTheList(ArrayList<Sale> sales) {
    sales.add(new Sale("Book", 10.95));
    sales.add(new Sale("Book", 30.00));
```

```
        sales.add(new Sale("Toy", 9.67));
        sales.add(new Sale("Book", 20.00));
    }
```

The `forEach` method in this ugly code does what its name suggests. It executes its lambda expression for every sale that comes its way. The ugly code uses `forEach` to tally up the prices of all book sales.

The output of this code is the same as the code in Listings 12-5 and 12-6. But there's more to writing code than getting the correct output. Figure 12-18 compares the three lambda expressions in the ugly code's stream.

Functional programming enthusiasts like the middle expression in Figure 12-18 — the `sale -> sale.price()` expression. That expression gets information from its parameter (`sale`) and returns a value (the `price` of that `sale`) to the rest of the code.

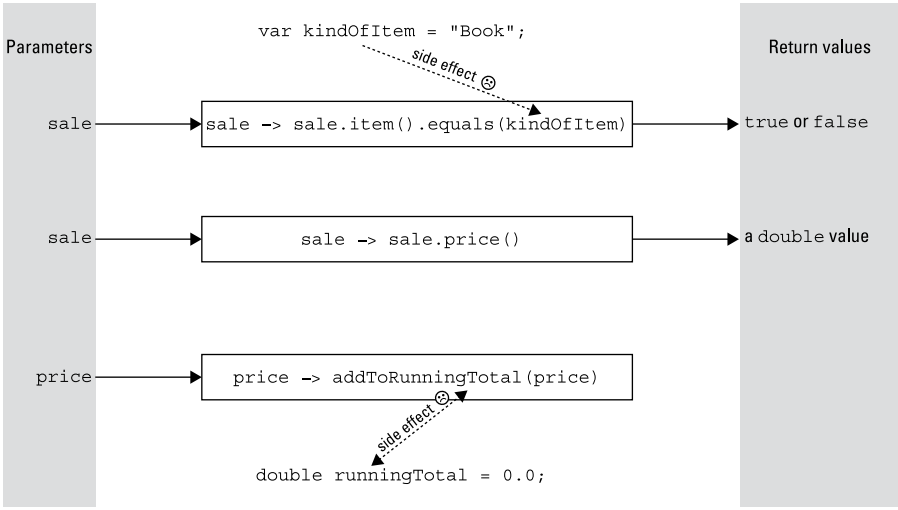


FIGURE 12-18: Functions with and without side effects.

Functional programmers don't like the other two expressions in Figure 12-18. That's because the other two expressions have side effects. A *side effect* takes place when a method or lambda expression refers to a value other than a parameter or its return value.

Side effects make programs difficult to understand. Programs without side effects have fewer bugs than programs with side effects. So, in functional programming, a side effect is considered a second-class citizen, a last resort, a tactic you use when you can't simply take in a parameter and return a result.

TRY IT OUT

You can always try the programming challenges that you dream up on your own. If you have no ideas to give you practice with functional programming, I have a couple of suggestions for you:

BE GENEROUS

Each employee has a name and a performance evaluation score. Find the total amount of money you'll pay in bonuses if you give a $100 bonus to each employee whose score is 3 or higher. In your code, make optimal use of lambda expressions and method references.

SAVE TIME

Each recipe has a name, a list of ingredients (some of which involve meat products), and an estimated preparation time. Find the average time estimate for cooking one of the vegetarian recipes.

INTERMEDIATE METHODS ARE JUST PLAIN LAZY

In the earlier section "The interpretation of streams," I warn that Java does nothing when you end a chain without a terminal operation. To double-check that I'm not lying about this, run the following code:

```
void main() {
    ArrayList<String> words =
            new ArrayList<>(Arrays.asList("I", "am", "happy."));

    var sentence = words.stream()
            .peek(word -> println(word))
            .reduce("", (word1, word2) -> word1 + " " + word2);

    println(sentence);
}
```

The lambda expression in Java's `peek` method never returns a value. So, aside from executing the lambda expression, `peek` doesn't modify the stream in any way. The `peek` method simply passes the stream's values on to the next method in line.

But `peek` performs an intermediate operation — not a terminal operation. So rerun the code with the following changes. You'll find out what happens when you end a chain with an intermediate operation:

```
var sentence = words.stream()
        .peek(word -> println(word));
        // .reduce("", (word1, word2) -> word1 + " " + word2);

// println(sentence);
```

STREAMLESS VARMINT

You don't need streams to make use of Java's lambda expressions. Run this code for a glimpse at the possibilities:

```
void main() {
    println(change("Hello", string -> {return string + "!";}));
}

String change(String str, Function<String, String> func) {
    return func.apply(str);
}
```

IN THIS CHAPTER

- Dealing with several values at a time
- Writing to disk files
- Running a program with different values each time

Chapter 13
Using Arrays to Juggle Values

Welcome to the Java Motel! No haughty bellhops, no overpriced room service, none of the silly jokes that I write in other chapters. Just a nice room at a darn good price!

Getting Your Ducks All in a Row

The Java Motel, with its ten comfortable rooms, sits in a quiet place off the main highway. Aside from a small, separate office, the motel is just one long row of ground-floor rooms. Each room is easily accessible from the spacious front parking lot.

Oddly enough, the motel's rooms are numbered 0 through 9. I could say that the numbering is a fluke — something to do with the builder's original design plan. But the truth is that starting with 0 makes the examples in this chapter easier to write. Computer programming structures start counting from 0 onward. That's all there is to it.

Anyway, you're trying to keep track of the number of guests in each room. To store all the numbers, you use an array.

An *array* is a row of values, like the row of rooms in a 1-floor motel. To picture the array, look at the Java Motel in Figure 13-1.

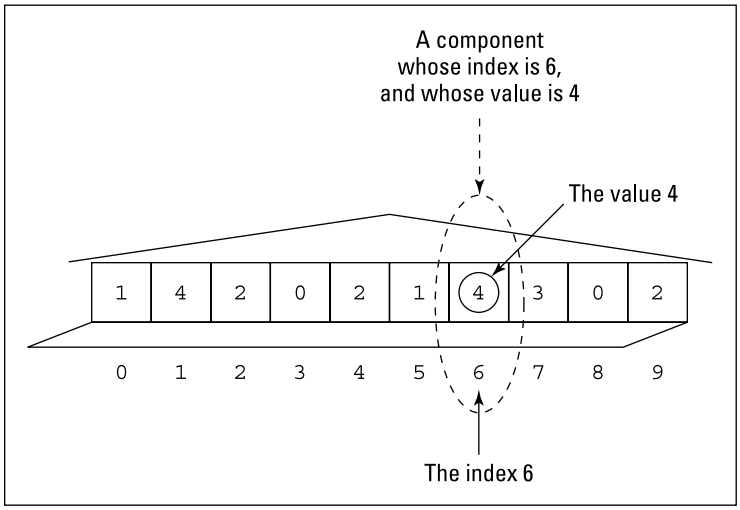


FIGURE 13-1: An abstract snapshot of rooms in the Java Motel.

In the lingo of this chapter, the entire row of rooms is called an *array.* Each room in the array is called a *component* of the array (also known as an array *element*). Each component has two numbers associated with it:

- The room number (a number from 0 to 9), which is called an *index* of the array
- A number of guests, which is a *value* stored in a component of the array

To declare an array with ten `int` values in it, you can write one fairly short statement:

```
int[] guests = new int[10];
```

If your declaration is inside of a method, you can use `var`, like so:

```
var guests = new int[10];
```

In some situations, it's handy to create an array in two separate steps, like this:

```
class MyClass {

    int[] guests;               // The var keyword isn't permitted here.

    void myMethod() {
        guests = new int[10];   // This must be inside a method body.
        // ...
    }

    // ...
}
```

In each of these code snippets, notice the use of the number 10. This number tells the computer to make the `guests` array have ten components. Each component of the array has a name of its own. The starting component is named *guests[0]*, the next is named *guests[1]*, and so on. The last of the ten components is named *guests[9]*.

REMEMBER

A declaration like

```
int[] guests;
```

doesn't reserve any space to store `int` values. It only reserves the variable name `guests` to eventually refer to an array of `int` values;

TIP

In an array declaration, you can put the square brackets before or after the variable name. In other words, you can write `int[] guests` or `int guests[]`. Java creates the same `guests` variable no matter which form you use.

How to book hotel guests

Business is booming at the Java Motel. (Most people prefer the Java Motel over its competitor — the C-Side Resort and Motor Lodge.) As the Java Motel's owner, manager, and system administrator, you have a file containing the number of guests in each room. A sample file named `GuestList.txt` is shown in Figure 13-2.

FIGURE 13-2: The `GuestList.txt` file.

```
1 4 2 0 2 1 4 3 0 2
```

After you've made a file, you can call on the `Scanner` class to get values from the file. The code is shown in Listing 13-1, and the resulting output is in Figure 13-3.

```
Room    Guests
0       1
1       4
2       2
3       0
4       2
5       1
6       4
7       3
8       0
9       2
```

FIGURE 13-3: Running the code from Listing 13-1.

CROSS REFERENCE

For tips on reading from disk files, refer to Chapter 9.

LISTING 13-1: **Filling an Array with Values**

```
import module java.base;

public class ShowGuests {

    public static void main(String[] args) throws IOException {
        int[] guests = new int[10];
        var diskScanner = new Scanner(new File("GuestList.txt"));

        for (int roomNum = 0; roomNum < 10; roomNum++) {
            guests[roomNum] = diskScanner.nextInt();
        }

        diskScanner.close();

        System.out.println("Room\tGuests");
        for (int roomNum = 0; roomNum < 10; roomNum++) {
            System.out.print(roomNum);
            System.out.print("\t");
            System.out.println(guests[roomNum]);
        }
    }
}
```

The code in Listing 13-1 has two `for` loops: The first loop reads numbers of guests, and the second loop writes numbers of guests.

Tab stops and other special things

In Listing 13-1, some calls to `print` and `println` use the `\t` escape sequence. It's called an *escape sequence* because you escape from displaying the letter `t` on the

screen. Instead, the characters `\t` stand for a tab. The computer moves forward to the next tab stop before printing any more characters. Java has a few of these handy escape sequences. Some of them are shown in Table 13-1.

TABLE 13-1: **Escape Sequences**

Sequence	Meaning
`\b`	Backspace
`\t`	Horizontal tab
`\n`	Line feed
`\f`	Form feed
`\r`	Carriage return
`\"`	Double quote "
`\'`	Single quote '
`\\`	Backslash \

Figure 13-3 looks good because I ran the program in a Mac Terminal window. In the Windows command prompt and Macintosh Terminal, tab stops are eight spaces apart. But in some environments, tab stops are only four spaces apart. With four-space stops, you don't see the sensible alignment that appears earlier, in Figure 13-3. If you want the columns to align no matter where your program runs, replace the tab characters with blank spaces in Listing 13-1. (Of course, if you're fussy, you have to check for numbers with more than one digit. But that's another story.)

Make life easy for yourself

Besides what you see in Listing 13-1, you have another way to fill an array in Java: with an *array initializer.* When you use an array initializer, you don't even have to tell the computer how many components the array has. The computer figures it out for you.

Here's another trick: You can also use Java's enhanced `for` loop (from Chapter 12) to step through an array. Listing 13-2 has an array initializer and an enhanced `for` loop.

LISTING 13-2: Short and Sweet Code!

```
void main() {
    int[] guests = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};

    println("Room\tGuests");

    int roomNum = 0;
    for (int numGuests : guests) {
        print(roomNum++);
        print("\t");
        println(numGuests);
    }
}
```

Listing 13-2 produces the same output as Listing 13-1, and Listing 13-2 is much easier to read. So why would anyone bother with the code in Listing 13-1? There are several reasons:

- **Putting data in a program, as I do in Listing 13-2, is seldom a good idea.**

  Time goes by. Hotel guests come and go. In this chapter's later "Writing to a file" section, you find out how to change the numbers in the `GuestList.txt` file. That's okay because people write to files all the time. But changing the numbers in Listing 13-2 would be another story. No one wants to edit a Java program whenever a hotel guest arrives or leaves.

- **Array initializers aren't versatile.**

  There's a good reason for calling them *initializers*. You can use this feature to initialize an array (as in Listing 13-2), but you can't use it to assign values to an existing array variable. For example, the initialization in Listing 13-2 is fine and dandy. But the following code is worthy of a slap on the wrist:

  ```
  // Bad code:
  int[] guests;
  guests = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};
  ```

- **An enhanced `for` loop, like the one in Listing 13-2, doesn't track the array's indices.**

  In Listing 13-1, both loops have a counter variable named `roomNum`. That makes it easy to print a room number on each line in the output. (Refer to Figure 13-3.)

  But an enhanced `for` loop doesn't have its own counter variable. To print the value of `roomNum` in Listing 13-2, you have to create your own `roomNum` variable and remember to update it each time through the loop. It's only a tiny

amount of extra work. But, in a complicated program, this extra work makes it easier to make a mistake. (I know. I've made those mistakes.)

» **The code in Listing 13-1 is a traditional Java `main` method. In comparison, the code in Listing 13-2 is a simple source file.**

That makes Listing 13-1 more verbose than Listing 13-2.

So, why should you have to wade through the `public class ShowGuests { public static` *`blah, blah ...`* business in Listing 13-1? One reason is that most Java main methods are written that way. In fact, before Java 21, all of Java's main methods were written that way. That's almost 30 years of Java code with lengthy, long-winded main methods. I can't write a Java book without highlighting some of those main methods.

TRY IT OUT

SPELL IT BACKWARD

Modify the code in Listing 13-1 so that it displays the rooms in reverse room-number order (starting with Room 9 and working down to Room 0).

MAKE YOUR MARKS

First, the user types five numbers on the keyboard. Then the program displays five lines of characters, each line having the same number of asterisks as one of the user's numbers. A run of the program might look like this:

```
5
3
9
0
10
*****
***
*********

**********
```

FIND THE BIGGEST

Add code to the end of Listing 13-1 so that the program finds the room(s) with the largest number of guests. The program's new output may look something like this:

```
4 guests in Room 1
4 guests in Room 6
```

To do this, have a variable named `largestSoFar`. Examine the array's elements one by one and then update the value of `largestSoFar` whenever you find a new, record-high number.

FIND THE TOTAL

Add code to the end of Listing 13-1 so that the program reports the total number of people staying at the motel. To do this, have a variable named `runningTotal`. Examine the array's values one by one, adding each such value to the big `runningTotal` value.

Array versus ArrayList: The Ultimate Showdown

This chapter's arrays are similar to the `ArrayList` collection in Chapter 12. So, if you need a structure that stores several values, which structure do you choose? An array or an `ArrayList`? It all depends on what you're trying to do.

» **Reasons for choosing an `ArrayList`:**

- An `ArrayList` can grow and shrink.
- After creating an `ArrayList` instance, you can add as many values to it as you want. The only limit is the amount of your Java application's memory. That's a very high limit.

  In contrast, every array has a fixed size. When you declare an array into existence, you tell Java how many elements the array has. You can't change your mind later in the program's code. After executing the line

  ```
  int[] guests = new int[10];
  ```

  in Listing 13-1, you can't put more than ten values into the `guests` array. Java simply won't allow it.
- An `ArrayList` has a rich set of helpful methods.

  For example, in Chapter 12, you insert `"paradox"` into position 2 of the `vocabulary` list by executing only one statement: `vocabulary.add(2, "paradox")`. But, to insert a value into an array, you need several loop iterations. (This chapter's later "Array Madness" section explains all of it.)

» **Reasons for choosing an array:**

- Arrays are more efficient.

The more you're able to do, the longer it takes to do anything at all. Sure, an `ArrayList` is more versatile than an array, but that (unfortunately) means operations on an `ArrayList` tend to be slower than the corresponding array operations. If you can live with having a fixed number of elements, an array may be the better choice.

- Arrays can store primitive types.

  Chapter 12 says that an `ArrayList` can't store a plain, old `int` value. Instead, you have to put big, fat instances of the `Integer` class in an `ArrayList`, like so:

```
ArrayList<Integer> list = new ArrayList<>();
```

  But the array in Listing 13-1 has no problem storing primitive `int` values:

```
int[] guests = new int[10];
```

  This can make an array less bulky than the corresponding `ArrayList`.

» **Other ways that an array differs from an `ArrayList`:**

- The notation for handling an array is different from the notation for handling `ArrayList` instances.

  In particular, arrays use square brackets — something you don't see with classes like `ArrayList`.

- Arrays are built into the Java programming language.

  The `ArrayList` class definition is part of Java's API, which (technically) is separate from the Java language itself. That's one reason some programs start with `import java.util.ArrayList`. In contrast, arrays are Java language features.

What makes `ArrayList` more versatile than an array? Under the hood, Java stores `ArrayList` values in a fixed-size array. When you execute code to add elements beyond the array's size, Java automatically creates a larger array and copies the old array's values into it. That's nice of Java. Isn't it?

Do You Have a Room?

You're sitting behind the desk at the Java Motel. Look! Here comes a party of five. These people want a room, so you need software that checks whether a room is vacant. If one is, the software modifies the `GuestList.txt` file (refer to Figure 13-2) by replacing the number 0 with the number 5. As luck would have it, the software is on your hard drive. The software is shown in Listing 13-3.

LISTING 13-3: Do You Have a Room?

```
void main() throws IOException {
    int[] guests = new int[10];
    int roomNum;

    var keyboard = new Scanner(System.in);
    var diskScanner = new Scanner(new File("GuestList.txt"));
    for (roomNum = 0; roomNum < 10; roomNum++) {
        guests[roomNum] = diskScanner.nextInt();
    }
    diskScanner.close();

    roomNum = 0;
    while (roomNum < 10 && guests[roomNum] != 0) {
        roomNum++;
    }

    if (roomNum == 10) {
        println("Sorry, no v cancy");
    } else {
        print("How many people for room ");
        print(roomNum);
        print("? ");

        guests[roomNum] = keyboard.nextInt();

        var listOut = new PrintStream("GuestList.txt");
        for (roomNum = 0; roomNum < 10; roomNum++) {
            listOut.print(guests[roomNum]);
            listOut.print(" ");
        }
        listOut.close();
    }
}
```

Figure 13-4 shows three runs of the code in Listing 13-3. The motel starts with two vacant rooms: Rooms 3 and 8. (Remember that the room numbers start with Room 0.) The first time you run the code, you put five people into Room 3. The second time you run the code, you put a party of ten in Room 8. (What a party!) The third time you run the code, you have no more vacant rooms. When the program discovers this, it displays the message `Sorry, no v cancy`, omitting at least one letter, in the tradition of all motel neon signs.

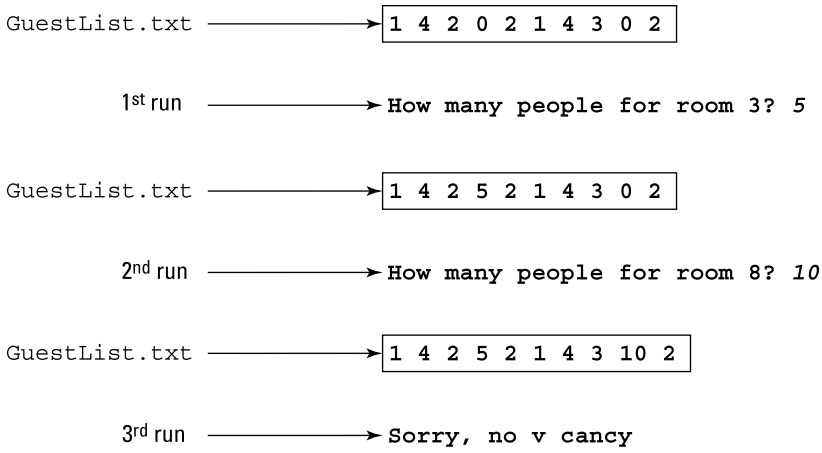

GuestList.txt
1 4 2 0 2 1 4 3 0 2
1st run
How many people for room 3? 5
GuestList.txt
1 4 2 5 2 1 4 3 0 2
2nd run
How many people for room 8? 10
GuestList.txt
1 4 2 5 2 1 4 3 10 2
3rd run
Sorry, no v cancy

FIGURE 13-4: Filling vacancies.

In Listing 13-3, the condition `roomNum < 10 && guests[roomNum] != 0` can be tricky. If you move things around and write **guests[roomNum] != 0 && roomNum < 10**, you can get yourself into lots of trouble. For details, see this book's website (`https://javafordummies.allmycode.com`).

TIP

Every array has a built-in `length` field. An array's *length* is the number of components in the array. For example, in Listing 13-3, you can replace the conditions in the `for` statements as follows:

```
for (roomNum = 0; roomNum < guests.length; roomNum++)
```

Writing to a file

The code in Listing 13-3 uses tricks from other chapters and sections of this book. The code's only brand-new feature is the use of `PrintStream` to write to a disk file. Think about any example in this book that calls `System.out.print` or `out.println` or their variants. What's *really* going on when you call one of these methods?

The thing called `System.out` is an object. The object is defined in the Java API. In fact, `System.out` is an instance of a class named `java.io.PrintStream` (or just `PrintStream`, to its close friends). Now each object created from the `PrintStream` class has methods named `print` and `println`. Just as each `Account` instance in Listing 7-4 (over in Chapter 7) has a `display` method, so the `PrintStream` instance named `out` has `print` and `println` methods. When you call `System.out.println`, you're calling a method that belongs to a `PrintStream` instance.

Okay, so what of it? Well, `System.out` always stands for some text area on your computer screen. If you create your own `PrintStream` object and you make that

object refer to a disk file, that `PrintStream` object refers to the disk file. When you call that object's `print` method, you write text to a file on your hard drive.

In Listing 13-3, when you say

```
var listOut = new PrintStream("GuestList.txt");

listOut.print(guests[roomNum]);
listOut.print(" ");
```

you're telling Java to write text to a file on your hard drive: the `GuestList.txt` file.

That's how you update the count of guests staying in the hotel. When you call `listOut.print` for the number of guests in Room 3, you may print the number 5. So, in Figure 13-4, a number in the `GuestList.txt` file changes from 0 to 5. Then in Figure 13-4, you run the program a second time. When the program gets data from the newly written `GuestList.txt` file, Room 3 is no longer vacant. This time, the program suggests Room 8.

TIP

This is more of an observation than a tip. Say that you want to *read* data from a file named `Employees.txt`. To do this, you make a scanner. You call `new Scanner(new File("Employees.txt"))`. If you accidentally call `new Scanner("Employees.txt")` without the `new File` part, the call doesn't connect to your `Employees.txt` file. But notice how you prepare to *write* data to a file. You make a `PrintStream` instance by calling `new PrintStream("GuestList.txt")`. You don't use `new File` anywhere in the call. If you goof and accidentally include `new File`, the Java compiler becomes angry, jumps out, and bites you.

What else can go wrong?

In any programming language, connecting to a file is a tricky business. You never know for sure whether the file exists. So, what happens when one of your constructors refers to a particular file? The answer depends partly on which constructor you call:

- **If you call `new Scanner(new File("GuestList.txt"))` and Java can't find the `GuestList.txt` file, Java says `FileNotFoundException`.**

  You can create workarounds for things like the `FileNotFoundException`. To read all about it, see Chapter 14.

- **If you call `new PrintStream("GuestList.txt")` and Java can't find a `GuestList.txt` file, Java makes a new `GuestList.txt` file on your computer's hard drive.**

The new file starts off being empty. It's up to you to put text (or some other kind of content) into it.

Here's something that may surprise you.

» **If you call** `new PrintStream("GuestList.txt")` **and Java *can* find a** `GuestList.txt` **file on your computer's hard drive, Java removes all content from that file.**

One way or another, after calling `new PrintStream("GuestList.txt")`, your hard drive has an empty `GuestList.txt` file.

Listing 13-3 reads from a `GuestList.txt` file and writes to the `GuestList.txt` file. It's important to make sure that these two operations don't collide with one another. Figure 13-5 shows what goes wrong if you call `new PrintStream("GuestList.txt")` before you've read all the values you need.

```
var diskScanner = new Scanner(new File("GuestList.txt"));
```
← Prepare to read from `GuestList.txt`.

```
var listout = new PrintStream("GuestList.txt");
```
← Prepare to write to `GuestList.txt` by clearing the file's contents.

```
for (roomNum = 0; roomNum < 10; roomNum++) {
    guests[roomNum] = diskScanner.nextInt();
}
```
← There's nothing to read because `GuestList.txt` contains no text.

FIGURE 13-5: Calling new PrintStream too soon.

In Listing 13-3, I'm careful to separate the reading of values in `GuestList.txt` from the writing of new values to `GuestList.txt`. Now you know why.

Array Madness

Near the start of this chapter, I write, ". . . to insert a value into an array, you need several loop iterations." This section's example (Listing 13-4) brings that point home.

LISTING 13-4: Squeezing in a Number

```
void main() {
    int[] scores = {75, 80, 87, 88, 90, 95, 99, -1, -1, -1};
    println(Arrays.toString(scores));

    for (int i = 6; i >= 2; i--) {
        scores[i + 1] = scores[i];
```

(continued)

LISTING 13-4: ***(continued)***

```
    }
    scores[2] = 85;

    println(Arrays.toString(scores));
}
```

A run of the program in Listing 13-4 is shown in Figure 13-6.

FIGURE 13-6: Inserting a number: Before and After.

```
[75, 80, 87, 88, 90, 95, 99, -1, -1, -1]
[75, 80, 85, 87, 88, 90, 95, 99, -1, -1]
```

The code in Listing 13-4 keeps track of test scores in my Java programming college course. I have ten students in the course. But, because I absolutely despise grading tests, I start the program in Listing 13-4 with only seven tests graded. In the array initializer, the last three `-1` values are placeholders. I'll overwrite these values when I finally muster the energy to grade the last three students' tests. Without these placeholders, I would never be able to grow the array beyond the first seven entries. (I could create a loop to copy the `scores` array's values into a larger array, but that would be cumbersome.)

In the middle of Listing 13-4, I drag my reluctant carcass to my desk and grade one more test. The eighth student scores an 85. I want to keep the array's values in increasing order. So, before inserting that 85 into the array, I have to move all the higher scores over by one position. That's what happens in the loop in Listing 13-4 — the loop that goes from index 6 down to index 2.

Moving five numbers from one position to another in an array is no big deal. But imagine an array with 945,847,523 elements. Inserting a value near the beginning of the array may require hundreds of millions of loop iterations. That's awful!

If you have to do lots of insertions into a bunch of values, you probably want to use Java's `LinkedList` collections class or something like it. For more information, visit `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/LinkedList.html`.

TIP

In Listing 13-4, Java's `Arrays.toString` method saves you from having to print array values one by one in a loop. That's quite convenient. But, for nicer-looking output, you probably don't want to use `Arrays.toString`. Instead, you go back to printing values one by one.

Return to Sanity

If you read the previous section (the one titled "Array Madness"), you know how much whining and complaining I do about grading tests. In this section, I quiet down and run another program to record test scores. This program (the one in Listing 13-5) illustrates a few more features of arrays.

LISTING 13-5: **A Few Tricks Using Arrays**

```
void main() {
    var keyboard = new Scanner(System.in);

    print("How many students? ");
    int arraySize = keyboard.nextInt();
    int[] scores = new int[arraySize];

    println("Enter " + arraySize + " students' scores:");
    for (int i = 0; i < arraySize; i++) {
        scores[i] = keyboard.nextInt();
    }

    Arrays.sort(scores);

    println(Arrays.toString(scores));
}
```

An array's size never changes. But that doesn't mean you have to know the array's size when you write the code. In Listing 13-5, the array's size is a variable whose value isn't known until a run of the program. In Figure 13-7, I set `arraySize` to 10.

```
How many students? 10
Enter 10 students' scores:
75 80 87 88 90 95 99
85
98 77
[75, 77, 80, 85, 87, 88, 90, 95, 98, 99]
```

FIGURE 13-7: Entering students' test scores.

The program prompts me for ten test scores. I type seven of them and press Enter. Then I stall for several hours. While the program waits patiently in a window on my computer screen, I drink some coffee, watch a few videos, and call a long-lost friend. After all that, I grade another student's test. I type the student's grade (85) and press Enter. Sometime the next day, I finally grade the last two students' tests. With much relief, I type the numbers 98 and 77 and then press Enter.

The program responds by calling the `Arrays` class's `sort` method. At the start of this chapter, I write that the `ArrayList` class has many more useful methods than Java's arrays have. That's true, but there's a `java.util.Arrays` class, and that class has some goodies of its own. For a look at some of these methods, visit `https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/util/Arrays.html`.

TECHNICAL STUFF

The `sort` method is a static member of the `Arrays` class. That's why you write `Arrays.sort` and not `scores.sort`.

TRY IT OUT

USING THE ARRAYS CLASS

Run this code to discover some useful methods in the `java.util.Arrays` package:

```
void main() {
    int[] guests = {1, 4, 2, 0, 2, 1, 4, 3, 0, 2};
    println(guests); // Not very useful!
    println(Arrays.toString(guests));

    int[] cSideGuests = new int[10];
    Arrays.fill(cSideGuests, 0);
    println(Arrays.toString(cSideGuests));

    Arrays.fill(cSideGuests, 5, 7, 2);
    println(Arrays.toString(cSideGuests));

    println(Arrays.equals(guests, cSideGuests));

    println(Arrays.binarySearch(guests, 3));

    Arrays.setAll(cSideGuests, i -> i + 1);
    println(Arrays.toString(cSideGuests));
}
```

TOO MANY WORDS

In a fit of shameless narcissism, I made a rough count of the number of words on pages 1–7 of this book's previous edition:

```
int[] words = {0, 296, 342, 405, 363, 350, 323, 101};
```

Write a program that prompts the user for a page number and responds with the count of words on that particular page.

TRACK YOUR WEIGHT

Initialize an array with your weight for each day in a week:

```
double[] weight = {145.7, 148.3, 147.2, 146.2, 147.0, 148.5, 146.9};
```

Write code to display a list of your gains and losses during the course of the week. For example, from the first day to the next, your weight change is `2.6`. By the following day, your change is `1.1`.

TALLY HO!

Add code to the end of Listing 13-1 so that the program reports the number of empty rooms, the number of rooms with only one guest, the number of rooms with exactly two guests, and so on. To do this, create an array named `howManyRoomsWith`. Examine the motel's rooms one by one. When you encounter an empty room, add 1 to `howManyRoomsWith[0]`. When you encounter a room with one guest, add 1 to `howManyRoomsWith[1]`. And so on.

Arrays of Objects

The Java Motel is open for business, now with improved guest registration software! The people who brought you this chapter's first section are always scratching their heads, looking for the best ways to improve their services. With some ideas from object-oriented programming, they've started thinking in terms of a `Room` class.

"And what," you ask, "would a `Room` instance look like?" That's easy. A `Room` instance has three properties: the number of guests in the room, the room rate, and a smoking/nonsmoking stamp. Figure 13-8 illustrates the situation.

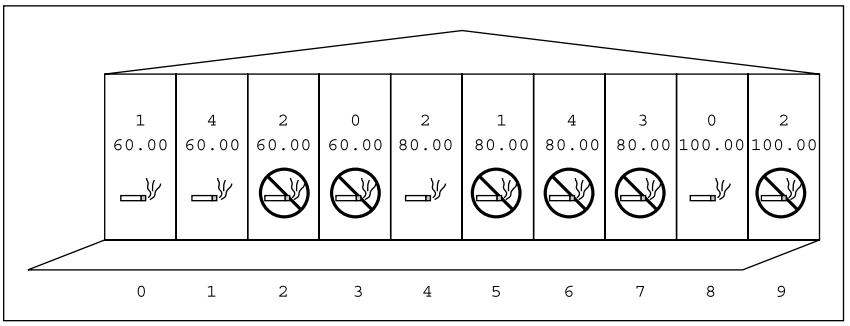


FIGURE 13-8: Another abstract snapshot of rooms in the Java Motel.

Listing 13-6 shows the code that describes the `Room` class. As promised, each instance of the `Room` class has three fields: the `guests`, `rate`, and `smoking` fields. (A `false` value for the `smoking` field indicates a nonsmoking room.) In addition, the entire `Room` class has a `static` field named `currency`. On my computer in the United States, this `currency` object makes room rates look like dollar amounts.

CROSS REFERENCE

To find out what *static* means, see Chapter 11. For information about the `currency` field, see Chapter 12.

LISTING 13-6: So This Is What a Room Looks Like!

```
package com.example.hotel;

import java.text.NumberFormat;
import java.util.Scanner;

import static java.lang.System.out;

public class Room {
    private int guests;
    private double rate;
    private boolean smoking;
    private static NumberFormat currency =
            NumberFormat.getCurrencyInstance();

    public void readRoom(Scanner diskScanner) {
        guests = diskScanner.nextInt();
        rate = diskScanner.nextDouble();
        smoking = diskScanner.nextBoolean();
    }

    public void writeRoom() {
        out.print(guests);
        out.print("\t");
        out.print(currency.format(rate));
        out.print("\t");
        out.println(smoking ? "yes" : "no");
    }
}
```

Listing 13-6 has a few interesting quirks, but I'd rather not describe them until after you see all the code in action. That's why, at this point, I move right on to the code that calls the Listing 13-6 code. After you read about arrays of rooms (shown in Listing 13-7), check out my description of the Listing 13-6 quirks.

WARNING

This warning is a deliberate repeat of an idea from Chapter 4, Chapter 7, and from who-knows-what-other chapter: *Be careful* when you use type `double` or type `float` to store money values. Calculations with `double` or `float` can be inaccurate. For more information (and more finger-wagging), see Chapters 4 and 7.

TIP

This tip has absolutely nothing to do with Java. If you're the kind of person who prefers a smoking room (with `boolean` field `smoking = true` in Listing 13-6), find someone you like — someone who can take off three consecutive days from work. Have that person sit with you and comfort you for 72 straight hours while you refrain from smoking. You might become temporarily insane while the nicotine leaves your body, but eventually you'll be okay. And your friend will feel like a real hero.

Using the Room class

Now you need an array of rooms. The code to create such a thing is in Listing 13-7. The code reads data from the `RoomList.txt` file. (Figure 13-9 shows the contents of the `RoomList.txt` file.)

```
1
60.00
true
4
60.00
true
2
60.00
false
0
60.00
false
2
80.00
true
1
80.00
false
4
80.00
false
3
80.00
false
0
100.00
true
2
100.00
false
```

FIGURE 13-9: A file of `Room` data.

Figure 13-10 shows a run of the code in Listing 13-7.

```
Room    Guests  Rate    Smoking?
0       1       $60.00  yes
1       4       $60.00  yes
2       2       $60.00  no
3       0       $60.00  no
4       2       $80.00  yes
5       1       $80.00  no
6       4       $80.00  no
7       3       $80.00  no
8       0       $100.00 yes
9       2       $100.00 no
```

FIGURE 13-10: What's happening at the Java Motel.

LISTING 13-7: **Would You Like to See a Room?**

```
import com.example.hotel.Room;

void main() throws IOException {

    Room[] rooms;
    rooms = new Room[10];
    var diskScanner = new Scanner(new File("RoomList.txt"));

    for (int roomNum = 0; roomNum < 10; roomNum++) {
        rooms[roomNum] = new Room();
        rooms[roomNum].readRoom(diskScanner);
    }

    println("Room\tGuests\tRate\tSmoking?");

    for (int roomNum = 0; roomNum < 10; roomNum++) {
        print(roomNum);
        print("\t");
        rooms[roomNum].writeRoom();
    }

    diskScanner.close();
}
```

Say what you want about the code in Listing 13-7. As far as I'm concerned, only one issue in the whole listing should concern you. And what, you ask, is that issue? Well, to create an array of *objects* — as opposed to an array made up of primitive values — you have to do three things: Make the array variable, make the array itself, and then construct each individual object in the array. This is different from creating an array of `int` values or an array containing any other primitive type

values. When you create an array of primitive type values, you do only the first two of these three things.

To help make sense of all this, follow along in Listing 13-7 and Figure 13-11 as you read the following points:

- `Room[] rooms;`: This declaration creates a `rooms` variable. This variable is destined to refer to an array (but doesn't yet refer to anything).
- `rooms = new Room[10];`: This statement reserves ten slots of storage in the computer's memory. The statement also makes the `rooms` variable refer to the group of storage slots. Each slot is destined to refer to an object (but doesn't yet refer to anything).
- `rooms[roomNum] = new Room();`: This statement is inside a `for` loop. The statement is executed once for each of the ten room numbers. For example, the first time through the loop, this statement says `rooms[0] = new Room()`. That first time around, the statement makes the slot `rooms[0]` refer to an actual object (an instance of the `Room` class).

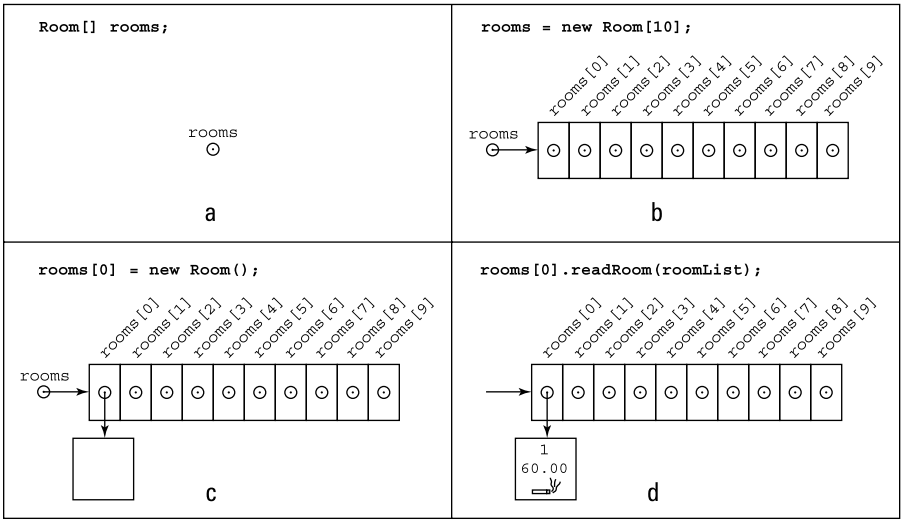


FIGURE 13-11: Steps in creating an array of objects.

Although it's technically not considered a step in array making, you still have to fill each object's fields with values. For instance, the first time through the loop, the `readRoom` call says `rooms[1].readRoom(diskScanner)`, which means, "Read data from the `RoomList.txt` file into the `rooms[1]` object's fields (the `guests`, `rate`, and `smoking` fields)." Each time through the loop, the program creates a new object and reads data into that new object's fields.

TIP

In Listing 13-7, I illustrate a point by separating parts in the creation of the `rooms` array. You don't have to do it this way. You can squeeze the steps together just as you do when creating arrays of primitive values. For instance, you can complete the first two steps in Listing 13-7 in one fell swoop, like this:

```
Room[] rooms = new Room[10];
```

REMEMBER

From Chapter 4 onward, I issue gentle warnings against using types such as `double` and `float` for storing currency values. For the most accurate currency calculations, use `int`, `long`, or — best of all — `BigDecimal`. You can read more about the dangers of the `double` type and currency values in Chapter 7.

The conditional operator

Listing 13-6 uses an interesting doodad called the *conditional operator*. This conditional operator takes three expressions and returns the value of just one of them. It's like a mini `if` statement. When you use the conditional operator, it looks something like this:

```
conditionToBeTested ? expression1 : expression2
```

The computer evaluates the *conditionToBeTested* condition. If the condition is true, the computer returns the value of *expression1*. But, if the condition is false, the computer returns the value of *expression2*.

So, in the code

```
smoking ? "yes" : "no"
```

the computer checks to see whether `smoking` has the value `true`. If so, the whole 3-part expression stands for the first string, `"yes"`. If not, the whole expression stands for the second string, `"no"`.

In Listing 13-6, the call to `println` causes either `"yes"` or `"no"` to display. Which string gets displayed depends on whether `smoking` has the value `true` or `false`.

TRY IT OUT

How do you learn Java? You learn it the same way you get to Carnegie Hall: Practice! Practice! Practice!

A CLASSY HOTEL

Supercharge this chapter's examples:

- Modify the code in Listing 13-3 so that it uses the `Room` class from Listing 13-6.
- Add to your work so that when you specify a number of incoming guests, you also enter the guests' preference for a smoking or nonsmoking room.
- Create a subclass of the `Room` class in Listing 13-6. In your subclass, each room has a maximum occupancy. Don't let your code put more guests in a room than the room's occupancy will allow.

DOUBLE YOUR PRESSURE

In Chapter 8, you create a `Student` class. Each student has a name and an ID number. For this programming challenge, imagine that each student has five grades — one for each of the five courses the student takes. Each grade is a `double` value from `0.0` to `4.0` (`4.0` is the best). A student's grade point average (GPA) is the average of the student's five grade values.

In this chapter's `Student` class, one of the fields is an array of five `double` values. Your program finds the student's GPA and displays it (along with the student's name and ID number) on the screen.

How to Argue with Your Code

Take a method header from Listing 13-6:

```
public void readRoom(Scanner diskScanner) {
```

This method has a parameter named `diskScanner`. The first time you call the method, you can make `diskScanner` read from the `RoomList.txt` file. The second time, you can make `diskScanner` read from the `wellNourishedKittens` file. The third time, `diskScanner` may represent the `underAppreciatedAuthors.md` file. This flexibility comes from the fact that `diskScanner` gets its value at the moment you call the method — and not a nanosecond before then.

The same kind of thing can be true of your program's `main` method. Every `main` method can have a `String[] args` parameter. When you launch a program, you can send one or more words to the `args` parameter. For example, if you send

```
Java For Dummies
```

to the program, then `args[0]` is `"Java"`, `args[1]` is `"For"`, and `args[2]` is `"Dummies"`. If you send

```
"Java For Dummies" Great!
```

to the program, then, because of the added quotation marks, `args[0]` is `"Java For Dummies"` and `args[1]` is `"Great!"`.

The way you send words to the `args` parameter depends on your IDE, but the general idea depends on two factors:

- **There are many ways to run a program.**

  Each of these ways is called a *run configuration*. Without knowing it, you've probably been using a default, one-size-fits-most run configuration to test this book's sample programs. That default configuration served you well.

  But now it's time for a change. A run configuration contains answers to several different questions: Which file contains the `main` method? Which version of Java will you run? Which of your drive's folders will be used? Where will the `System.out` text appear? And so on.

- **One of these answers is named *program arguments*.**

  Most IDEs have some kind of Modify Run Configuration menu option. When you select this option, you see a dialog box with places for all these answers. In that dialog box, one text field is labeled Program Arguments. Your IDE takes whatever you type in that text field and sends it to `args` when your code runs.

  You can change a configuration's program arguments before each run or create several different configurations, each with its own program arguments. For help setting up program arguments in Eclipse and IntelliJ IDEA, visit this book's website (`https://javafordummies.allmycode.com`).

Settling the argument

Listing 13-8 shows you how to use program arguments in your code.

LISTING 13-8: Generate a File of Numbers

```
void main(String[] args) throws IOException {

    var generator = new Random();

    if (args.length < 2) {
        println("Usage: MakeRandomNumsFile filename number");
```

```
            System.exit(1);
        }

        var printOut = new PrintStream(args[0]);
        int numLines = Integer.parseInt(args[1]);

        for (int count = 1; count <= numLines; count++) {
            printOut.println(generator.nextInt(10) + 1);
        }

        printOut.close();
    }
```

At the top of Figure 13-12, I create a configuration for a run of Listing 13-8. Before I run the program, I set the program arguments to `RandomNumbers.txt 5`.

FIGURE 13-12: A run of the code in Listing 13-8.

WARNING

The look of the configuration dialog box depends on which IDE you use, and which version of that IDE. Your IDE's configuration dialog box may look different from the one in Figure 13-12.

When I run the program, the configuration in Figure 13-12 makes the value of `args[0]` be `"RandomNumbers.txt"` and the value of `args[1]` be `"5"`. So the program's assignment statements end up having the following meaning:

```
var printOut = new PrintStream("RandomNumbers.txt");
int numLines = Integer.parseInt("5");
```

The program creates a file named `RandomNumbers.txt` and sets `numLines` to `5`. So, later in the code, the program randomly generates five values and puts those values into `RandomNumbers.txt`. (Refer to the bottom of Figure 13-12.)

Notice how each program argument in Listing 13-8 is a `String` value. When you look at `args[1]`, you don't see the number 5 — you see the string `"5"` with a digit character in it. Unfortunately, you can't use that `"5"` to do any counting. To get an `int` value from `"5"`, you have to apply the `parseInt` method. (Again, see Listing 13-8.)

You don't need an IDE to feed arguments to a program. Imagine that you have a Java class named `ThreeLittleWords`. Depending on your setup, you may be able to open a Command Prompt or Terminal window and type `java ThreeLittleWords.java Java For Dummies`. If all goes well, the `args` array will contain the words `Java For Dummies`.

Checking for the right number of program arguments

What happens if the user makes a mistake? What if the user omits the number `5` in the text field in Figure 13-12?

Then the computer assigns `"RandomNumbers.txt"` to `args[0]` but assigns nothing to `args[1]`. This is bad. If the computer ever reaches the statement

```
int numLines = Integer.parseInt(args[1]);
```

the program crashes with an unfriendly `ArrayIndexOutOfBoundsException`.

What do you do about this? In Listing 13-8, you check the length of the `args` array. You compare `args.length` with `2`. If the `args` array has fewer than two components, you display a message on the screen and exit from the program. Figure 13-13 shows the resulting output.

```
Usage: MakeRandomNumsFile filename number
```

FIGURE 13-13: The code in Listing 13-8 tells you how to run it.

WARNING

Despite the checking of `args.length` in Listing 13-8, the code still isn't crash-proof. If the user types `five` instead of `5`, the program takes a nosedive with a `NumberFormatException`. The program's `for` loop can't go from the `int` value `1` to the `String` value `"five"`. So this program's second `args` value can't be a word. It has to be a number (and a whole number, at that). I can add `if` statements to Listing 13-8 to make the code more bulletproof, but checking for the `NumberFormatException` is better done in Chapter 14.

The sun is about to set on this book's discussion of arrays. But before you leave the subject of arrays, think about this: An array is a row of things, and not every kind of thing fits into just one row. Take the first few examples in this chapter involving the motel. The motel rooms, numbered 0 through 9, are in one long line. But what if you move up in the world? You buy a big hotel with 50 floors and with 100 rooms on each floor. Then the data starts looking square-shaped. You have 50 rows, and each row contains 100 items. Sure, you can think of the rooms as though they're all in one long row, but why should you have to do that? How about having a 2-dimensional array? It's a square-shaped array in which each component has two indices: a row number and a column number. Alas, I have no space in this book to show you a 2-dimensional array (and I can't afford a big hotel's prices, anyway). But if you visit this book's website (`https://javafordummies.allmycode.com`), you can read all about it.

TRY IT OUT

You can never get too much practice:

MINE IS THE BIGGEST

Write a program whose program arguments include three `int` values. As its output, the program displays the largest of the three `int` values.

IN THIS CHAPTER

- Recovering from bad input and other nasty situations
- Making your code (more or less) crash-proof
- Defining your own exception class

Chapter 14
Looking Good When Things Take Unexpected Turns

September 9, 1945: A moth flies into one of the relays of the Harvard Mark II computer and gums up the works. This becomes the first recorded case of a real-life computer bug.

April 19, 1957: Herbert Bright, manager of the data processing center at Westinghouse in Pittsburgh, receives an unmarked deck of computer punch cards in the mail (which is like receiving an unlabeled thumb drive in the mail today). Mr. Bright guesses that this deck comes from the development team for FORTRAN — the first computer programming language. He's been waiting a few years for this software. (No web downloads were available at the time.)

Armed with nothing but this good guess, Bright writes a small FORTRAN program and tries to compile it on his IBM 704. (The IBM 704 lives in its own, specially built, 2,000-square-foot room. With vacuum tubes instead of transistors, the machine has a whopping 32K of RAM. The operating system has to be loaded from tape before the running of each program, and a typical program takes between two and four hours to run.) After the usual waiting time, Bright's attempt to

compile a FORTRAN program comes back with a single error: a missing comma in one of the statements. Bright corrects the error, and the program runs like a charm.

July 22, 1962: Mariner I, the first US spacecraft aimed at another planet, is destroyed when it behaves badly 4 minutes after launch. The bad behavior is attributed to a missing bar (like a hyphen) in the formula for the rocket's velocity.

Around the same time, orbit computation software at NASA is found to contain the incorrect statement `DO 10 I=1.10` (instead of the correct `DO 10 I=1,10`). In modern notation, this is like writing `do10i = 1.10` in place of `for (int i=1; i<=10; i++)`. The change from a comma to a period turns a loop into an assignment statement.

January 1, 2000: Experts believe that computer programs will mistake `01/01/00` for January 1, 1900. Prophets of doom predict worldwide disasters and unparalleled destruction. For most systems, this Y2K problem causes negligible damage. Most of the doomsayers are rightfully embarrassed.

Any historically accurate facts in these notes were borrowed from the following sources: the Computer Folklore newsgroup (`https://groups.google.com/forum/#!forum/alt.folklore.computers`), the Free On-line Dictionary of Computing (`https://foldoc.org`), *Computer* magazine (`www.computer.org/computer-magazine`), and other web pages of the IEEE (`www.computer.org`). All inaccuracies stem from this author's lunatic musings.

Garbage In

You're taking inventory. This means counting item after item, box after box, and marking the numbers of such things on log sheets, in little handheld gizmos, and into forms on computer keyboards. A particular part of the project involves entering the number of boxes you find on the shelf labeled Big Dusty Boxes That Haven't Been Opened Since Year One. Rather than break the company's decades-old habit, you decide not to open any of these boxes. You arbitrarily assign the value $3.25 to each box.

Listing 14-1 shows the software to handle this bit of inventory. The software has a flaw, which is revealed in Figure 14-1. When the user enters a whole number value, things are okay. But when the user enters something else (like the number 3.5), the program comes crashing to the ground. Surely something can be done about this. Computers are stupid, but they're not so stupid that they should fail royally whenever a user enters an improper value.

LISTING 14-1: Counting Boxes

```
import java.text.NumberFormat;
import java.util.Scanner;

import static java.lang.System.out;

public class InventoryA {

    public static void main(String[] args) {
        final double boxPrice = 3.25;
        var keyboard = new Scanner(System.in);
        var currency = NumberFormat.getCurrencyInstance();

        out.print("How many boxes do we have? ");
        String numBoxesIn = keyboard.next();
        int numBoxes = Integer.parseInt(numBoxesIn);

        out.print("The value is ");
        out.println(currency.format(numBoxes * boxPrice));
    }
}
```

```
How many boxes do we have? 3
The value is 59.75



How many boxes do we have? 3.5
Exception in thread "main" java.lang.NumberFormatException Create breakpoint : For input string: "3.5"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:588)
    at java.base/java.lang.Integer.parseInt(Integer.java:685)
    at Inventorya.main(InventoryA.java:15)



How many boxes do we have? three
Exception in thread "main" java.lang.NumberFormatException Create breakpoint : For input string: "three"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:588)
    at java.base/java.lang.Integer.parseInt(Integer.java:685)
    at InventoryA.main(InventoryA.java:15)
```

FIGURE 14-1: Three separate runs of the code in Listing 14-1.

The key to fixing a program bug is examining the message that appears when the program crashes. The inventory program's message says `java.lang.NumberFormatException`. That means a class named *NumberFormatException* is in the `java.lang` API package. Somehow, the call to `Integer.parseInt` brought this `NumberFormatException` class out of hiding.

Well, here's what's going on. The Java programming language has a mechanism called exception handling. With *exception handling*, a program can detect that something is about to go wrong and respond by creating a brand-new object. In the official terminology, the program is said to be *throwing* an exception. That new object, an instance of the `Exception` class, is passed like a hot potato from one piece of code to another until some piece of code decides to *catch* the exception. When the exception is caught, the program executes some recovery code, buries the exception, and moves on to the next normal statement as though nothing had ever happened. The process is illustrated in Figure 14-2.

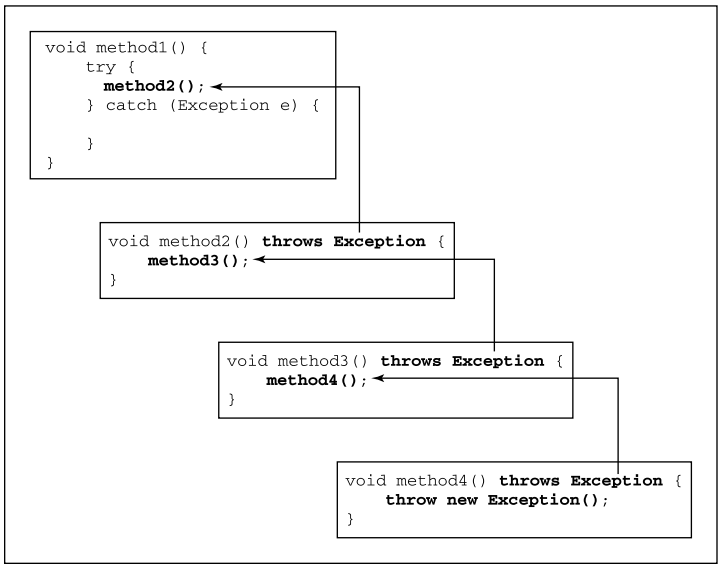


FIGURE 14-2: Throwing, passing, and catching an exception.

The whole thing is done with the aid of several Java keywords. These keywords are described in this list:

- `throw`: Creates a new exception object.
- `throws`: Indicates that a method may pass the exception up to whatever code called the method.
- `try`: Encloses code that has the potential to create a new exception object. In the usual scenario, the code inside a `try` clause contains calls to methods whose code can create one or more exceptions.
- `catch`: Deals with the exception, buries it, and then moves on.

So the truth is out. By some chain of events like the one shown in Figure 14-2, the method `Integer.parseInt` can throw a `NumberFormatException`. When you call `Integer.parseInt` with `3.5` or `three`, Java generates a `NumberFormatException` and passes it up the chain of method calls. The passing continues until some Java code catches the exception. But, in Listing 14-1, there's no code to catch a `NumberFormatException`. So, when you input `3.5` or `three`, the program crashes and you get embarrassing messages like the ones in Figure 14-1.

TIP

The Java application programming interface (API) documentation for the `parseInt` method says, "Throws: `NumberFormatException` — if the string does not contain a parsable integer." Once in a while, reading the documentation actually pays off.

SOME THINGS NEVER CHANGE

The main method in Listing 14-1 begins with this statement:

```
final double boxPrice = 3.25;
```

In this declaration of the `boxPrice` variable, take special note of the word `final`. This Java keyword indicates that the value of `boxPrice` can't be changed. If I add another assignment statement to Listing 14-1, like

```
boxPrice = 3.50;
```

or

```
boxPrice += 1.0;
```

the compiler barks back at me with the message `cannot assign a value to final variable 'boxprice'`.

When I write the code in Listing 14-1, the use of the `final` keyword isn't absolutely necessary. But the `final` keyword provides some extra protection. When I initialize `boxPrice` to `3.25`, I intend to use this same number just as it is, over and over again. I know darn well that I don't intend to change the `boxPrice` variable's value. Of course, in a 10,000-line program, I can become confused and try to assign a new value to `boxPrice` somewhere deep down in the code. To prevent me (or anyone else) from accidentally changing the `boxPrice` value, I declare the `boxPrice` variable to be final. It's just good, safe programming practice.

Making things right

The code in Listing 14-1 may throw an exception. You'd better catch the exception to keep the program from crashing. Listing 14-2 shows the catching of an exception.

LISTING 14-2: A Safer Way to Count Boxes

```
void main() {
    final double boxPrice = 3.25;
    var currency = NumberFormat.getCurrencyInstance();

    String numBoxesIn = readln("How many boxes do we have? ");

    try {
        int numBoxes = Integer.parseInt(numBoxesIn);
        print("The value is ");
        println(currency.format(numBoxes * boxPrice));
    } catch (NumberFormatException e) {
        print(e.getMessage());
        println(" ... Cannot interpret the input.");
    }
    println("Have a nice day!");
}
```

Figure 14-3 shows three runs of the code from Listing 14-2. When a misguided user types **three** instead of **3**, the program maintains its cool by displaying `For input string: "three" . . . Cannot interpret the input`. The trick is to enclose the call to `Integer.parseInt` inside a `try` clause. If you do this, the computer watches for exceptions when any statement inside the `try` clause is executed. If an exception is thrown, the computer jumps from inside the `try` clause to a `catch` clause below it. In Listing 14-2, the computer jumps directly to the `catch (NumberFormatException e)` clause. The computer executes the two `print` statements inside the clause and then marches on with normal processing. In Listing 14-2, this "normal processing" means printing `Have a nice day!`.

An entire `try-catch` assembly — complete with a `try` clause, `catch` clause, and what-have-you — is called a *try statement.* Sometimes, for emphasis, I call it a *try-catch statement.*

The parameter in a catch clause

Take a look at the `catch` clause in Listing 14-2 and pay particular attention to the words `(NumberFormatException e)`. This looks a lot like a method's parameter

list, doesn't it? In fact, every `catch` clause is like a little mini-method with its own parameter list.

```
How many boxes do we have? 3
The value is $9.75
Have a nice day!

How many boxes do we have? three
For input string: "three" . . . Cannot interpret the input.
Have a nice day!

How many boxes do we have? -25
The value is -$81.25
Have a nice day!
```

FIGURE 14-3: Three runs of the code in Listing 14-2.

The parameter list always has an exception type name and then a parameter. The parameter is an object — an instance of the `NumberFormatException` class. When an exception is caught, Java makes the `catch` clause's parameter refer to that exception object. In other words, the name *e* in Listing 14-2 stores a bunch of information about the exception. To take advantage of this, you can call some of the exception object's methods.

In Listing 14-2, I call the exception's `getMessage` method. Lo and behold! The `getMessage` call returns the text `For input string: "three"`. That's helpful information. Java hands this information to the `e` variable whenever it can't make sense of the user's `three` input.

In addition to the `getMessage` method, each exception has a handy `printStackTrace` method. To see how this works, rewrite the `catch` clause in Listing 14-2 as follows:

```
} catch (NumberFormatException e) {
    e.printStackTrace();
}
```

With this new `catch` clause, a run of the `inventory` program may look like the run shown in Figure 14-4. When you call `printStackTrace`, you see a list of methods that were running when the exception was thrown, along with line numbers in each of the methods. In Figure 14-4, the display includes `Integer.parseInt` and the `main` method.

```
How many boxes do we have? three
Have a nice day!
java.lang.NumberFormatException Create breakpoint : For input string: "three"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:588)
    at java.base/java.lang.Integer.parseInt(Integer.java:685)
    at InventoryBWithStackTrace.main(InventoryBWithStackTrace.java:8)
```

FIGURE 14-4: Java's stack trace.

Notice the similarity between the run in Figure 14-4 and two of the runs in Figure 14-1.

- **In Figure 14-4:** Java throws a `NumberFormatException`. The code catches the exception and calls a method that prints a stack trace. Then the program keeps running (to print `Have a nice day`, for example).
- **In Figure 14-1:** Java throws a `NumberFormatException`. None of the code in Listing 14-1 catches the exception, so Java does what it always does for any uncaught exceptions: It prints a stack trace and then abruptly ends the run of the program. In other words, your program crashes. Nothing else gets calculated, printed, or processed in any way.

TECHNICAL STUFF

In Figure 14-4, notice that `Have a nice day` appears before the exception's stack trace. That's peculiar. Doesn't Java call `e.printStackTrace()` before it prints `Have a nice day`? It does. But the stack trace and `nice day` outputs follow different paths to get to your computer screen. So they don't always appear in the expected order.

The `print` and `println` calls in Listing 14-2 go to your old friend, `System.out`. But `printStackTrace` doesn't write directly to `System.out`. The `printStackTrace` method writes to something called `System.err`, which, by default, shows up in the same place as `System.out`. When you mix `System.out.println` calls with `printStackTrace` calls, the order in which Java displays the information isn't predictable. There are ways to tweak the code so that `System.out` and `System.err` go to different places on your screen, but that's not normally something you'd be anxious to do.

TRY IT OUT

I can't think of a clever way to connect the *Try* in *TryItOut* with the `try` in `try-catch` statements. If you think of something, scribble it in the margin on this page. Then try this little challenge:

DON'T BE A QUITTER

Add `try-catch` statements to keep the following code from crashing:

```
void main() {
    var words = new String[5];
```

```
    int i = 0;
    do {
        words[i] = readln("");
    } while (!words[i++].equals("Quit"));

    for (int j = 0; j < 5; j++) {
        println(words[j].length());
    }
}
```

Do it yourself

In the previous sections, you confuse your inventory program by typing `three` instead of `3`. Nice work! What else can go wrong today? Are there other kinds of exceptions — ones that have nothing to do with the `NumberFormatException`? Sure, plenty of different exception types are out there. You can even create one of your own. You wanna try? If so, look at Listings 14-3 and 14-4.

LISTING 14-3: Making Your Own Kind of Exception

```
package com.example.inventory;

public class OutOfRangeException extends RuntimeException {
    public OutOfRangeException(String message) {
        super("A value is out of range.\n" + message);
    }
}
```

LISTING 14-4: Using Your Custom-Made Exception

```
import com.example.inventory.OutOfRangeException;

void main() {
    final double boxPrice = 3.25;
    var currency = NumberFormat.getCurrencyInstance();

    String numBoxesIn = readln("How many boxes do we have? ");

    try {
        int numBoxes = Integer.parseInt(numBoxesIn);
        if (numBoxes < 0) {
            throw new OutOfRangeException("You typed " + numBoxes +
```

(continued)

LISTING 14-4: ***(continued)***

```
                    ". There's no such thing as a negative box.");
        }
        print("The value is ");
        println(currency.format(numBoxes * boxPrice));
    } catch (NumberFormatException e) {
        print(e.getMessage());
        println(" ... Cannot interpret the input.");
    } catch (OutOfRangeException e) {
        println(e.getMessage());
    }
}
```

Listings 14-3 and 14-4 remedy a problem that quietly cropped up earlier. Look at the last of the three runs in Figure 14-3. The user reports that the shelves have −25 boxes, and the computer takes this value without blinking an eye. The truth is that you would need a black hole (or some other exotic, space-time-warping phenomenon) to have a negative number of boxes on any shelf in your warehouse. So the program should become upset if the user enters a negative number of boxes, which is what the code in Listing 14-4 does. To see what I mean, look at Figure 14-5.

```
How many boxes do we have? 3
The value is $9.75

How many boxes do we have? three
For input string: "three" . . . Cannot interpret the input.

How many boxes do we have? -25
A value is out of range.
You typed -25. There's no such thing as a negative box.
```

FIGURE 14-5: Three runs of the code from Listings 14-3 and 14-4.

The code in Listing 14-3 declares a new kind of exception class: `OutOfRangeException`. In many situations, typing a negative number would work just fine, so `OutOfRangeException` isn't built-in to the Java API. However, in the inventory program, a negative number of boxes should be flagged as an anomaly.

How do you know that the class in Listing 14-3 is an exception of some kind? You know because the class declaration's first line says `extends RuntimeException`. The class in Listing 14-3 is a particular kind of exception called a *runtime exception*. To learn more about runtime exceptions, see the section "The Buck Stops Here, Except When It Doesn't," later in this chapter.

TECHNICAL STUFF

When you call

```
throw new OutOfRangeException("You typed " + numBoxes +
        ". There's no such thing as a negative box.");
```

in Listing 14-4, you pass the string `"You typed " + numBoxes + . . .` to the `message` parameter in Listing 14-3. The constructor in Listing 14-3 doesn't do much on its own with that `message` value. Instead, the constructor combines the message with its own `"A value is out of range.\n"` string. Then the constructor uses `super` to pass the combined string on to its superclass — the `RuntimeException` class. By the time you're done, the call `e.getMessage()` gives you a whole bunch of text: `A value is out of range . . . blah, blah, . . . no such thing . . . blah blah.`

CROSS REFERENCE

To read more about Java's `super` keyword, refer to Chapter 10.

Who will catch the exception?

Take one more look at Listing 14-4. Notice that more than one `catch` clause can accompany a single `try` clause. When an exception is thrown inside a `try` clause, Java starts reviewing the accompanying list of `catch` clauses. It starts at whatever `catch` clause comes immediately after the `try` clause and works its way down the program's text.

For each `catch` clause, Java asks itself, "Is the exception that was just thrown an instance of the class in this clause's parameter list?"

- If not, Java skips this `catch` clause and moves on to the next `catch` clause in line.
- If so, Java executes this `catch` clause and then skips past all other `catch` clauses that come with this `try` clause. Java goes on and executes whatever statements come after the whole `try`–`catch` statement.

For some concrete examples, see Listings 14-5 and 14-6.

LISTING 14-5: Yet Another Exception

```
package com.example.inventory;

public class NumberTooLargeException extends OutOfRangeException {
    public NumberTooLargeException(String message) {
        super("A value is too large.\n" + message);
    }
}
```

LISTING 14-6: Catch Me If You Can

```
import com.example.inventory.*;

void main() {
    final double boxPrice = 3.25;
    final int maxBoxes = 1000;
    NumberFormat currency = NumberFormat.getCurrencyInstance();

    String numBoxesIn = readln("How many boxes do we have? ");

    try {
        int numBoxes = Integer.parseInt(numBoxesIn);

        if (numBoxes < 0) {
            throw new OutOfRangeException("You typed " + numBoxes +
                    ". There's no such thing as a negative box.");
        }
        if (numBoxes > maxBoxes) {
            throw new NumberTooLargeException(numBoxes +
                    " is larger than the maximum of " + maxBoxes);
        }
        print("The value is ");
        println(currency.format(numBoxes * boxPrice));

    } catch (NumberFormatException e) {
        print(e.getMessage());
        println(" ... Cannot interpret the input.");

    } catch (OutOfRangeException e) {
        println(e.getMessage());

    } catch (Exception e) {
        println(e.getMessage());
    }

    println("That's that.");
}
```

To run the code in Listings 14-5 and 14-6, you need one additional Java program file. You need the `OutOfRangeException` class in Listing 14-3.

Listing 14-6 addresses the scenario in which you have limited shelf space. You don't have room for more than 1,000 boxes, but once in a while the program asks how many boxes you have, and somebody enters the number *1001* accidentally. In cases like this, Listing 14-6 performs a quick reality check: Any number of boxes over 1,000 is tossed out as being unrealistic.

Listing 14-6 watches for a `NumberTooLargeException`, but to make life more interesting, Listing 14-6 has no `catch` clause for the `NumberTooLargeException`. In spite of this, everything still works out just fine. It's fine because `NumberTooLargeException` is declared to be a subclass of `OutOfRangeException`, and Listing 14-6 has a `catch` clause for the `OutOfRangeException`.

You see, because the `NumberTooLargeException` class is a subclass of the `OutOfRangeException` class, any instance of `NumberTooLargeException` is just a special kind of `OutOfRangeException`. So, in Listing 14-6, Java may start looking for a clause to catch a `NumberTooLargeException`. When Java stumbles upon the `OutOfRangeException` catch clause, Java says, "Okay, I've found a match. I'll execute the statements in this `catch` clause."

To keep from having to write this whole story over and over again, I introduce some new terminology. I say that the `catch` clause with parameter `OutOfRangeException` *matches* the `NumberTooLargeException` that's been thrown. I call this `catch` clause a *matching catch clause.*

The following list describes different things the user may do and how the computer responds. As you read, you can follow along by looking at the runs shown in Figure 14-6:

- **The user enters an ordinary whole number, like the number *3.*** All statements in the `try` clause are executed. Then Java skips past all the `catch` clauses and executes the code that comes immediately after all the `catch` clauses. (See Figure 14-7.)
- **The user enters something that's not a whole number, like the word *fish.*** The code throws a `NumberFormatException`. Java skips past the remaining statements in the `try` clause. Java executes the statements inside the first `catch` clause — the clause with parameter `NumberFormatException`. Then Java skips past the second and third `catch` clauses and executes the code that comes immediately after all the `catch` clauses. (See Figure 14-8.)
- **The user enters a negative number, like the number –25.** The code throws an `OutOfRangeException`. Java skips past the remaining statements in the `try` clause. Java even skips past the statements in the first `catch` clause. (After all, an `OutOfRangeException` isn't any kind of a `NumberFormatException`. The `catch` clause with parameter `NumberFormatException` isn't a match for this `OutOfRangeException`.) Java executes the statements inside the second `catch` clause — the clause whose parameter is of type `OutOfRangeException`. Then Java skips past the third `catch` clause and executes the code that comes immediately after all the `catch` clauses. (See Figure 14-9.)

```
How many boxes do we have? 3
The value is $9.75
That's that.

How many boxes do we have? fish
For input string: "fish" . . . Cannot interpret the input.
That's that.

How many boxes do we have? -25
A value is out of range.
You typed -25. There's no such thing as a negative box.
That's that.

How many boxes do we have? 1001
A value is out of range.
A value is too large.
1001 is larger than the maximum of 1000
That's that.
```

FIGURE 14-6: Four runs of the code from Listing 14-6.

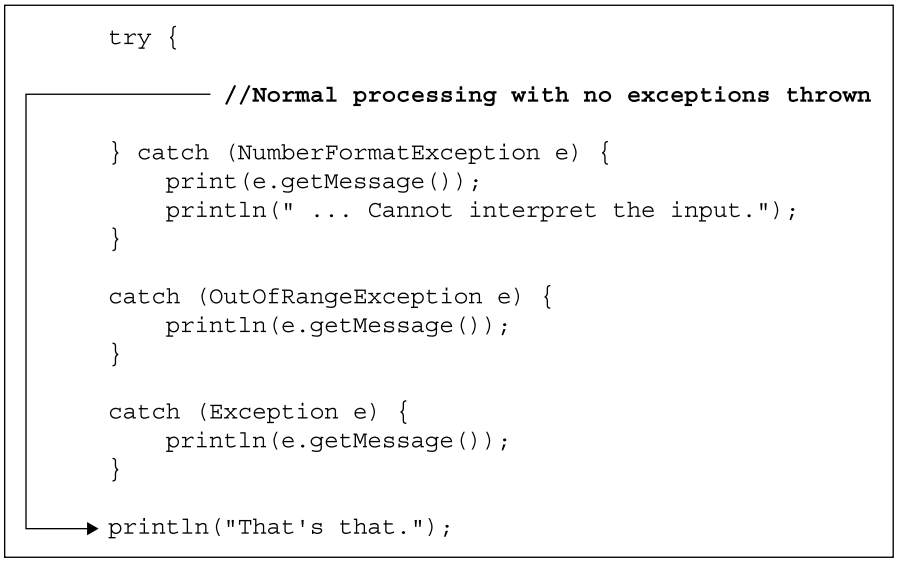


FIGURE 14-7: No exception is thrown.

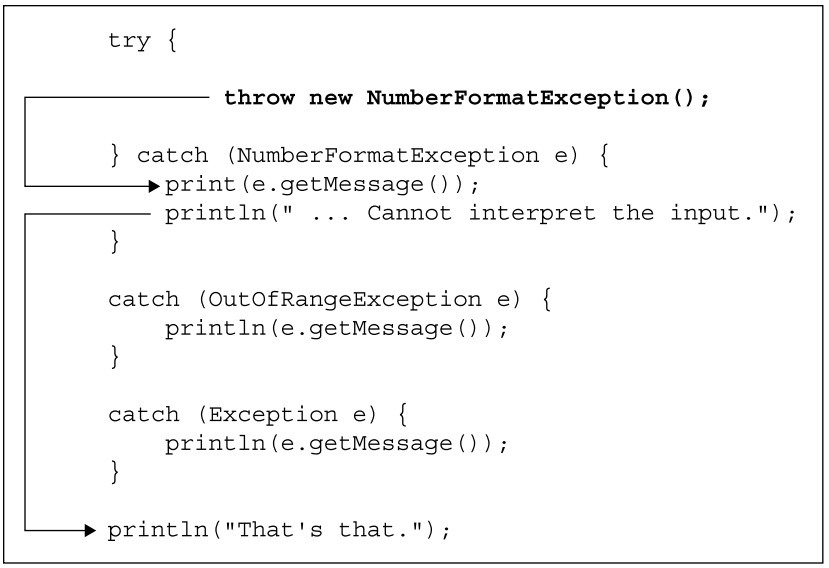


FIGURE 14-8: Java throws a `NumberFormatException`.

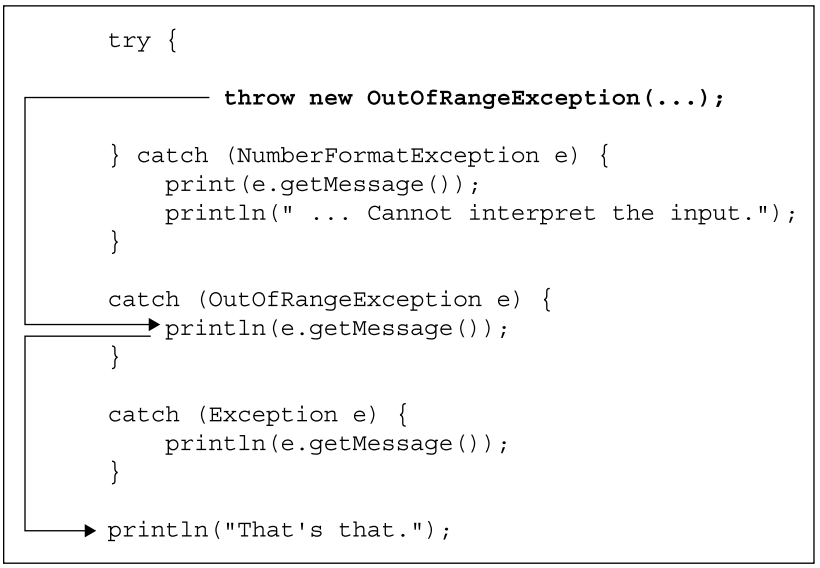


FIGURE 14-9: Java throws an `OutOfRangeException`.

- **The user enters an unrealistically large number, like the number 1001.** The code throws a `NumberTooLargeException`. Java skips past the remaining statements in the `try` clause. Java even skips past the statements in the first `catch` clause. (After all, a `NumberTooLargeException` isn't any kind of `NumberFormatException`.)

  But, according to the code in Listing 14-5, `NumberTooLargeException` is a subclass of `OutOfRangeException`. When Java reaches the second `catch` clause, Java says, "Hmm! A `NumberTooLargeException` is a kind of `OutOfRangeException`. I'll execute the statements in this `catch` clause — the clause with parameter of type `OutOfRangeException`." It's a match.

  Java executes the statements inside the second `catch` clause. Then Java skips the third `catch` clause and executes the code that comes immediately after all the `catch` clauses. (See Figure 14-10.)

- **Something else, something quite unpredictable, happens. (I don't know what.)** With my unending urge to experiment, I reached into the `try` clause of Listing 14-6 and added a statement that commits a serious crime. The statement divides a number by zero:

  ```
  try {
      out.println(1 / 0);
      int numBoxes = Integer.parseInt(numBoxesIn);
  ```

  When Java encounters this statement, it throws an `ArithmeticException`. Java skips past the remaining statements in the `try` clause. Then Java skips past the statements in the first and second `catch` clauses. When Java reaches the third `catch` clause, I can hear Java say, "Hmm! An `ArithmeticException` is a kind of `Exception`. I've found a matching `catch` clause — a clause with a parameter of type `Exception`. I'll execute the statements in this `catch` clause."

  So Java executes the statements inside the third `catch` clause. Then Java executes the code that comes immediately after all the `catch` clauses. (See Figure 14-11.)

When Java looks for a matching `catch` clause, Java latches on to the topmost clause that fits one of the following descriptions:

- The clause's parameter type is the same as the type of the exception that was thrown.
- The clause's parameter type is a superclass of the exception's type.

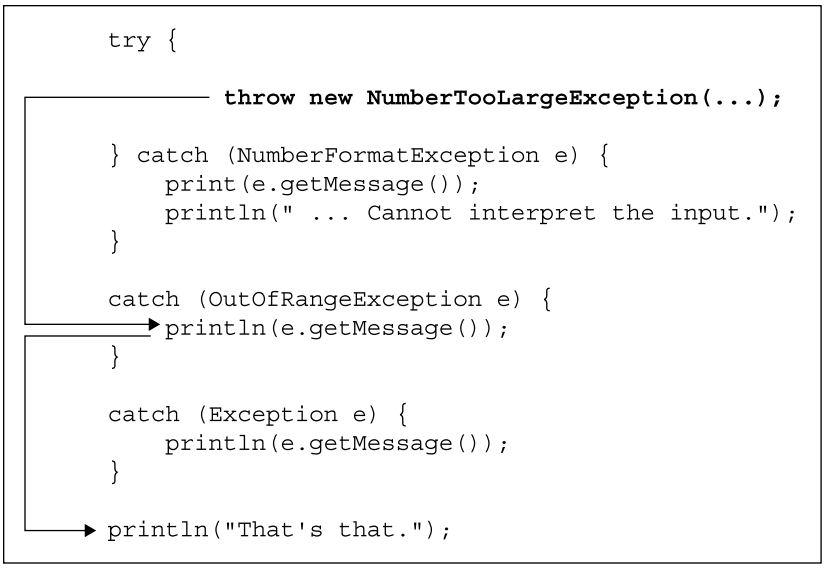


FIGURE 14-10: Java throws a `NumberTooLargeException`.

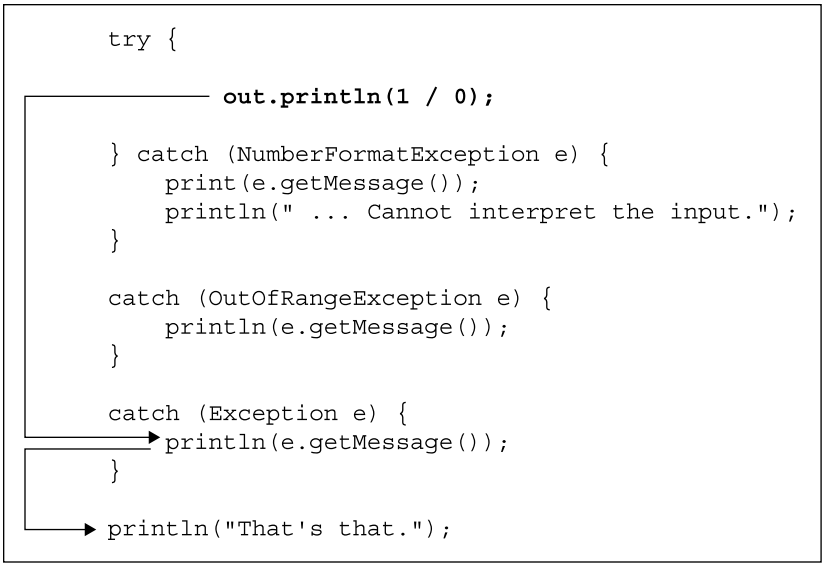


FIGURE 14-11: Java throws an `ArithmeticException`.

If a better match appears farther down the list of `catch` clauses, that's just too bad. Imagine that you added a `catch` clause with a parameter of type `NumberTooLargeException` to the code in Listing 14-6. Imagine also that you put this new `catch` clause *after* the `catch` clause with a parameter of type `OutOfRangeException`. Then, because `NumberTooLargeException` is a subclass of the `OutOfRangeException` class, the code in the new `NumberTooLargeException` clause would never be executed. That's just the way the cookie crumbles.

Catching two or more exceptions at a time

You can catch more than one kind of exception in a single `catch` clause. For example, in a particular inventory program, you might want the same text to accompany a `NumberFormatException` and your own `OutOfRangeException`. In that case, you can rewrite part of Listing 14-6 this way:

```
} catch (NumberFormatException | OutOfRangeException e) {
    println(e.getMessage());
    println("Input a number from 0 to " + maxBoxes + ".");
} catch (Exception e) {
    println(e.getMessage());
}
```

The pipe symbol, `|`, tells Java to catch either a `NumberFormatException` or an `OutOfRangeException`. If you throw an exception of either type, the program adds `Input a number from 0 to 1000` to its helpful output. If you throw an exception that's neither a `NumberFormatException` nor an `OutOfRangeException`, the program jumps to the last `catch` clause where you don't get that `Input a number . . .` output.

TECHNICAL STUFF

Imagine that a user runs Listing 14-6 and types the word `three` instead of the number `3`. You'd be tempted to call that an error. Right? Well, Java calls that a `NumberFormatException`. It's an `Exception`, not an `Error`. Java has things called *errors*, but they're not the same as this chapter's exceptions. The basic difference is that an exception is something your program may be able to deal with. Your program can catch the exception, respond appropriately, and then move on with its processing life. On the other hand, an *error* is something so heinous that your program should stop in its tracks. For example, an `OutOfMemoryError` occurs when your program runs out of space to store values. If that happens, there's no sense in trying to catch the error and recover. Your program terminates, and it's "back to the drawing board."

TRY IT OUT

Try your hand at this coding task:

LOOSEN UP A BIT

In Listing 14-6, `boxPrice` and `maxBoxes` have fixed values. Make improvements to the code so that the user enters both those values. Remember that some values for these quantities make no sense. For example, a negative number of boxes is never too many boxes. Use `try-catch` statements to handle inappropriate user input.

The Buck Stops Here, Except When It Doesn't

Listing 14-7 has a scaled-down version of an example from Chapter 9. The code tries to read values from a disk file and then write some results to the screen, but the code doesn't work.

LISTING 14-7: Not Writing Payroll Checks

```
/*
* This code does not compile.
*/

void main() {
    println("Starting payroll ... ");
    cutCheck();
    println("Payroll completed.");
}

void cutCheck() {
    var diskScanner = new Scanner(new File("EmployeePayment.txt"));
    String name = diskScanner.nextLine();
    double amountPaid = diskScanner.nextDouble();

    print("Pay to the order of " + name);
    println(String.format(" ***$%,.2f", amountPaid));

    diskScanner.close();
}
```

When you construct a `Scanner` with `System.out`, nothing much can go wrong. But when you construct a `Scanner` with a `File`, you have to be careful. Maybe you misspelled the filename (`EnployeePaymant.txt`). Maybe your file is in the cloud and the file is unreadable because the network is down. Maybe you lack permission to access *any* files. Who knows?

Java doesn't let you call `new Scanner(new File(`*anything_at_all*`))` unless you acknowledge that there's some risk. The problem is that the code deep inside the `Scanner` constructor's body can throw an exception. This kind of exception is an instance of the `FileNotFoundException` class. When you try to compile the code in Listing 14-7, you see an unwelcome message such as

```
unreported exception java.io.FileNotFoundException;
        must be caught or declared to be thrown
```

The Java programming language has two kinds of exceptions. They're called *checked* and *unchecked* exceptions:

- The potential throwing of a checked exception must be acknowledged in the code.
- The potential throwing of an unchecked exception doesn't need to be acknowledged in the code.

A `FileNotFoundException` is one of Java's checked exception types. When you call a constructor or method that has the potential to throw a `FileNotFoundException`, you need to acknowledge that exception in the code.

Now, when I say that an exception is *acknowledged in the code*, what do I really mean?

```
/*
 * The programmer wishes to thank the FileNotFoundException,
 * without which this code could not have been written.
 */
```

No, that's not what it means to be acknowledged in the code. Acknowledging an exception in the code means one of two things:

- The statements (including any constructor or method calls) that can throw the exception are inside a `try` clause. That `try` clause has a `catch` clause with a matching exception type in its parameter list.
- The statements (including any constructor or method calls) that can throw the exception are inside a method that has a `throws` clause in its header. The `throws` clause contains a matching exception type.

If you're confused by the wording of these two bullets, don't worry. The next two listings illustrate the points made in the bullets.

Catch it soon

In Listing 14-8, the `new Scanner` call is inside a `try` clause. That `try` clause has a `catch` clause with exception type `FileNotFoundException`.

LISTING 14-8: Acknowledging with a try-catch Statement

```
void main() {
    println("Starting payroll ... ");
    cutCheck();
    println("Payroll completed.");
}

void cutCheck() {
    try {
        var diskScanner = new Scanner(new File("EmployeePayment.txt"));
        String name = diskScanner.nextLine();
        double amountPaid = diskScanner.nextDouble();

        print("Pay to the order of " + name);
        println(String.format(" ***$%,.2f", amountPaid));

        diskScanner.close();
    } catch (FileNotFoundException e) {
        println(e.getMessage());
    }
}
```

Figure 14-12 shows two runs of the code in Listing 14-8. For the first run, an `EmployeePayment.txt` file contains the lines `Barry Burd` and `5000.00`. For the second run, the user's hard drive contains no `EmployeePayment.txt` file.

```
Starting payroll . . .
Pay to the order of Barry Burd ***$5,000.00
Payroll completed.

Starting payroll . . .
EmployeePayment.txt (No such file or directory)
Payroll completed.
```

FIGURE 14-12: With and without an `EmployeePayment.txt` file.

When I look at Listing 14-8, my heart is filled with pride. I've acknowledged the `FileNotFoundException`, prevented the code from printing `Pay to the order of`, and avoided a nasty program crash. In fact, I handled the exception as soon as it arose. I put out that fire in the `cutCheck` method before it could spread and burn up the `main` method.

But wait! Did I do the right thing? Here's an important principle in software design: Don't be too hasty to catch an exception. Looking back at Figure 14-12, the

second run ends with the words `Payroll completed`. Is that true? Will Barry actually receive the $5000.00 that he so rightfully deserves? No. That second run in Figure 14-12 throws a `FileNotFoundException`. The `cutCheck` method in Listing 14-8 catches the exception and doesn't tell the `main` method that anything went wrong. So the `main` method, gleeful in its ignorance of the `FileNotFoundException`, prints `Payroll completed`.

I shouldn't have shielded the code's main method from the troublesome `FileNotFoundException`. Instead, I should have relayed the exception from the `cutCheck` method up to the `main` method. The example in the next section shows you how.

Catch it later

In Listing 14-9, the `cutCheck` method says "This `FileNotFoundException` isn't my problem. Let `main` deal with it."

LISTING 14-9: A Method Admits Its Shortcomings

```
void main() {
    println("Starting payroll ... ");
    try {
        cutCheck();
        println("Payroll completed.");
    } catch (FileNotFoundException e) {
        println(e.getMessage());
    }
}

void cutCheck() throws FileNotFoundException {
    var diskScanner = new Scanner(new File("EmployeePayment.txt"));
    String name = diskScanner.nextLine();
    double amountPaid = diskScanner.nextDouble();

    print("Pay to the order of " + name);
    println(String.format(" ***$%,.2f", amountPaid));

    diskScanner.close();
}
```

Figure 14-13 shows two runs of the code in Listing 14-9. For the first run, an `EmployeePayment.txt` file contains the lines `Barry Burd` and `5000.00`, and the run ends with the words `Payroll completed`. For the second run, the user's hard drive contains no `EmployeePayment.txt` file, and the program doesn't display those misleading words.

```
Starting payroll . . .
Pay to the order of Barry Burd ***$5,000.00
Payroll completed.

Starting payroll . . .
EmployeePayment.txt (No such file or directory)
```

FIGURE 14-13: Payroll completed? Maybe yes, maybe no.

The important part of Listing 14-9 is in the `cutCheck` method's header. That header ends with `throws FileNotFoundException`. By announcing that it throws a `FileNotFoundException`, method `cutCheck` passes the buck. What this `throws` clause really says is, "I realize that a statement inside this method has the potential to throw a `FileNotFoundException`, but I'm not acknowledging the exception in a `try-catch` statement. Java compiler, please don't bug me about this. Instead of having a `try-catch` statement, I'm passing the responsibility for acknowledging the exception to the `main` method (the method that called the `cutCheck` method)."

Indeed, in the `main` method, the call to `cutCheck` is inside a `try` clause. That `try` clause has a `catch` clause with a parameter of type `FileNotFoundException`. So everything is okay. Method `cutCheck` passes the responsibility to the `main` method, and the `main` method accepts the responsibility with an appropriate `try-catch` statement. Everybody's happy. Even the Java compiler is happy.

To better understand the `throws` clause, imagine a volleyball game in which the volleyball is an exception. When a player on the other team serves, that player is throwing the exception. The ball crosses the net and heads directly toward you. If you pound the ball back across the net, you're catching the exception. But if you pass the ball to another player, you're using the `throws` clause. In essence, you're saying, "Here, other player — *you* deal with this exception."

REMEMBER

A statement in a method can throw an exception that's not matched by a `catch` clause. This includes situations in which the statement throwing the exception isn't even inside a `try` block. When this happens, execution of the program jumps out of the method that contains the offending statement. Execution jumps back to whatever code called the method in the first place.

TIP

A method can name more than one exception type in its `throws` clause. Just use commas to separate the names of the exception types, as in the following example:

```
throws FileNotFoundException, NumberFormatException, ArithmeticException
```

Checked or unchecked?

Sneak a peek at Listing 14-3 and notice that your newly declared `OutOfRangeException` class extends Java's `RuntimeException` class. Why would you want to extend `RuntimeException`? Here's the story:

- **Any subclass of `RuntimeException` is unchecked.**

  In fact, any descendant (sub-subclass, sub-sub-subclass, and so on) of `RuntimeException` is unchecked. The unchecked exceptions represent errors that can crop up almost anywhere in your program. If your code crashes because of an unchecked exception, you should think about rewriting the code.

  Some common unchecked exceptions include the `NumberFormatException` (of Listings 14-2, 14-4, and others), the `ArithmeticException`, the `IndexOutOfBoundsException`, the infamous `NullPointerException`, and many others. When you write Java code, much of your code is susceptible to these exceptions, but enclosing the code in `try` clauses (or passing the buck with `throws` clauses) is completely optional.

- **Any exception that's not a descendant of `RuntimeException` is checked.**

  Checked exceptions represent challenges that the outside world may impose on your program. Like it or not, some files simply aren't where they should be, so you had better acknowledge this in your program and write code to make the best of it. Java's checked exceptions include the `FileNotFoundException`, (See Listings 14-8 and 14-9.) the `PrinterException`, the `SQLException`, and a gang of other interesting exceptions.

READER'S ACKNOWLEDGMENTS

TRY IT OUT

The following code doesn't compile, because the code throws an unacknowledged `FileNotFoundException`:

```
// BAD CODE:
void main() {
    var diskScanner = new Scanner(new File("numbers.txt"));

    int[] numerators = new int[5];
    int[] denominators = new int[5];

    int i = 0;

    while (diskScanner.hasNextInt()) {
        numerators[i] = diskScanner.nextInt();
```

```
        denominators[i] = diskScanner.nextInt();
        i++;
    }

    for (int j = 0; j < numerators.length; j++) {
        println(numerators[j] / denominators[j]);
    }

    diskScanner.close();
}
```

Fix the unacknowledged `FileNotFoundException` so that the code compiles. Then notice that, depending on the values in the `numbers.txt` file, some other exceptions may be thrown during a run of the program. Add one or more `try-catch` statements to display messages about these exceptions without letting the program crash.

IF IT'S BROKEN, FIX IT

Add `try-catch` statements or `throws` clauses (or a mixture of these two things) to fix the following broken code:

```
// BAD CODE:
void main() {
    var fileIn = new File("input");
    var fileInStrm = new FileInputStream(fileIn);
    var dataInStrm = new DataInputStream(fileInStrm);

    var fileOut = new File("output");
    var fileOutStrm = new FileOutputStream(fileOut);
    var dataOutStrm = new DataOutputStream(fileOutStrm);

    int numFilesCopied = 0;

    try {
        while (true) {
            dataOutStrm.writeByte(dataInStrm.readByte());
        }
    } catch (EOFException e) {
        numFilesCopied = 1;
    }
}
```

When you've gotten the code to compile, create a file named `input` and run the code to see whether it creates the file named `output`.

Try, Try Again!

Once upon a time, I was a young fellow living with my parents in Philadelphia and just starting to drive a car. I was heading toward a friend's house and thinking about who-knows-what when another car came from nowhere and bashed my car's passenger door. This kind of thing is called a RunARedLightException.

Anyway, both cars were still drivable, and we were squat in the middle of a busy intersection. To avoid causing a traffic jam, we both pulled over to the nearest curb. I fumbled for my driver's license and opened the door to get out of my car.

And that's when the second accident happened. As I was getting out of my car, a city bus was coming by. The bus hit me and rolled me against my car a few times. This kind of thing is called a DealWithLawyersException.

The truth is that everything came out just fine. I was bruised but not battered. My parents paid for the damage to the car, so I never suffered any financial consequences. (I managed to pass on the financial burden by putting the RunARedLightException into my `throws` clause.)

This incident helps to explain why I think the way I do about exception handling. In particular, I wonder, "What happens if, while the computer is recovering from one exception, a second exception is thrown?" After all, the statements inside a `catch` clause aren't immune to calamities.

Well, the answer to this question is anything but simple. For starters, you can put a `try` statement inside a `catch` clause. This protects you against unexpected, potentially embarrassing incidents that can crop up during the execution of the `catch` clause. But when you start worrying about cascading exceptions, you open up a slimy can of worms. The number of scenarios is large, and things can become complicated quickly. The program in Listing 14-10 helps you sort things out.

LISTING 14-10: Using Two Files

```
void main() {
    println("Starting payroll ... ");
    try {
        cutCheck();
        println("Payroll completed.");
```

```
    } catch (FileNotFoundException e) {
        println(e.getMessage());
    }
}

void cutCheck() throws FileNotFoundException {
    var diskScanner = new Scanner(new File("EmployeePayment.txt"));
    var diskScanner2 = new Scanner(new File("LegalInfo.txt"));

    String name = diskScanner.nextLine();
    double amountPaid = diskScanner.nextDouble();
    String disclaimer = diskScanner2.nextLine();

    print("Pay to the order of " + name);
    println(String.format(" ***$%,.2f", amountPaid));
    println(disclaimer);

    diskScanner.close();
    diskScanner2.close();
}
```

The output of a run from Listing 14-10 might look like the text in Figure 14-14.

```
Starting payroll . . .
Pay to the order of Barry Burd ***$5,000.00
We're not responsible for anything at all.
Payroll completed.
```

FIGURE 14-14: You'll hear from our lawyers!

But what happens if, after successfully opening the first file (`EmployeePayment.txt`), Java can't find the second file (`LegalInfo.txt`)? Then Java throws a `FileNotFoundException` and jumps immediately out of the `cutCheck` method. Java never calls `diskScanner.close()`, and the connection to `EmployeePayment.txt` lives on in limbo. With a small program like the one in Listing 14-10, this is no big deal. But if your code repeatedly fails to close such resources, you could be in big trouble.

That's why Java has a try-with-resources feature. A *try-with-resources statement* keeps track of all the things you've opened and closes them automatically when they're no longer needed. Listing 14-11 has the full story.

LISTING 14-11: Making Sure to Close Resources

```
void main() {
    println("Starting payroll ... ");
    try {
        cutCheck();
        println("Payroll completed.");
    } catch (FileNotFoundException e) {
        println(e.getMessage());
    }
}

void cutCheck() throws FileNotFoundException {
    try (var diskScanner = new Scanner(new File("EmployeePayment.txt"));
         var diskScanner2 = new Scanner(new File("LegalInfo.txt"))) {

        String name = diskScanner.nextLine();
        double amountPaid = diskScanner.nextDouble();
        String disclaimer = diskScanner2.nextLine();

        print("Pay to the order of " + name);
        println(String.format(" ***$%,.2f", amountPaid));
        println(disclaimer);

    } catch (FileNotFoundException e) {
        println("Abnormal return from the cutCheck method ... ");
        throw e;
    }
}
```

In Listing 14-11, the declarations of `diskScanner1` and `diskScanner2` are in parentheses after the word `try`. The parenthesized declarations tell Java to close `diskScanner1` and `diskScanner2` automatically after execution of the statements in the `try` clause. Java closes these connections whether you throw an exception or not.

Look at all the statements in the body of the `cutCheck` method. Not one of these statements contains the word `close`. You don't need any explicit `close` calls. The try-with-resources statement does all the closing for you.

You can access all kinds of resources (files, databases, connections to servers, and others) and have peace of mind knowing that Java will sever the connections automatically when necessary. Life is good!

Before we bid a fond farewell to Listing 14-11, let's take a moment to look at the `cutCheck` method's `catch` clause. As in Listing 14-9, we want the code's `main`

method to do something about a missing `LegalInfo.txt` file. We could add lots of clunky `if` statements to make `main` bend to our will, but it's better to have `main` respond to the original `FileNotFoundException`. In the `cutCheck` method's `catch` clause, the variable `e` stands for that unsavory `FileNotFoundException`. So the statement `throw e` does the trick. That statement says, "Pass this despicable `FileNotFoundException` on to whichever method called me." And the "method that called me" is `main`.

TIP

In a try-with-resources statement, the `catch` clause is optional. In Listing 14-11, if you remove the `catch` clause, the program behaves almost the same way. The only thing you lose is the possibility of getting the `Abnormal return from the cutCheck method` output.

DON'T MISS A TRICK

TRY IT OUT

Modify the code in Listing 14-11 so that it checks for sensible values in the two disk files. Does the `EmployeePayment.txt` file have a name and a double value? Does the `LegalInfo.txt` file contain any text?

IN THIS CHAPTER

- Writing and using a Java interface
- Working with abstract classes

Chapter 15
Fancy Reference Types

In previous chapters, you may have read about the things that full-time and part-time employees have in common. In particular, both the `FullTimeEmployee` and `PartTimeEmployee` classes can extend the `Employee` class. That's nice to know if you're running a small business, but what if you're not running a business? What if you're taking care of house pets?

This chapter explores the care of house pets and other burning issues.

Java's Types

Chapter 4 explains that Java has these two kinds of types:

- **Primitive types:** Java has a total of eight primitive types: The four you use most often are `int`, `double`, `boolean`, and `char`.
- **Reference types:** Java's API has thousands of reference types, and, when you write a Java program, you define new reference types.

  Java's `String` type is a reference type. So are Java's `Scanner`, `Random`, `ArrayList`, and `File` types. In Chapter 10, you create your own `Employee`, `FullTimeEmployee`, and `PartTimeEmployee` reference types. When you write

```
public class NiceExample {
    public static void main(String[] args) {
```

you define a reference type named `NiceExample`. When you put

```
void main() {
```

in a file named `EvenBetterExample.java`, you implicitly define a reference type named `EvenBetterExample`. You may not realize it, but every array belongs to a reference type. The list of reference types goes on and on.

In Java, reference types are everywhere. But until this point in the book, the only reference types you see are classes and arrays. Java has other kinds of reference types, and this chapter explores the possibilities.

The Java Interface

Think about a class (such as an `Employee` class) and a subclass (such as a `FullTimeEmployee` class). The relationship between a class and its subclass is one of inheritance. In many real-life families, a child inherits assets from a parent. And in Chapter 10, the `FullTimeEmployee` class inherits `name` and `jobTitle` fields from the `Employee` class. That's the way it works.

But consider the relationship between an editor and an author. The editor says, "By signing this contract, you agree to submit a completed manuscript by the ninth of September." Despite any excuses that the author gives before the deadline date (and, believe me, authors make plenty of excuses), the relationship between the editor and the author is one of obligation. The author agrees to take on certain responsibilities; and, in order to continue being an author, the author must fulfill those responsibilities. (By the way, there's no subtext in this paragraph — none at all.)

Now consider Barry Burd. Who? Barry Burd — that guy who writes *Java For Dummies* and certain other *For Dummies* books (all from Wiley Publishing). He's a college professor, and he's also an author. You want to mirror this situation in a Java program, but Java doesn't support multiple inheritance. You can't make Barry extend both a `Professor` class and an `Author` class at the same time.

Fortunately for Barry, Java has interfaces. An interface is a kind of reference type. In fact, the code to create an interface looks a lot like the code to create a class:

```
public interface MyInterfaceName {
    // blah, blah, blah
}
```

An interface is a lot like a class, but an interface is different. (What else is new? A cow is like a planet, but it's quite a bit different. Cows moo; planets hang in space.)

Anyway, when you read the word *interface*, you can start by thinking of a class. Then, in your head, note that

- A class can extend only one parent class, but a class can implement many interfaces.
- A parent class is a bunch of stuff that a class inherits. But an *interface* is a bunch of stuff that an implementing class is *obligated to provide*.

What about poor Barry? He can be an instance of a `Person` class with all the fields that any person has — `name`, `address`, `age`, `height`, `weight`, and so on. He can also implement more than one interface:

- Because Barry implements a `Professor` interface, he must have methods named `teachStudents`, `adviseStudents`, and `gradePapers`.
- Because he implements an `Author` interface, he must have methods named `writeChapters`, `reviewChapters`, `answerEmail`, and so on.

Declaring two interfaces

Imagine two different kinds of data. One is a column of numbers that comes from an array. Another is a table (with rows and columns) that comes from a disk file. What might these two things have in common?

I don't know about you, but I may want to display both kinds of data. So, I can write code to create a contract. The contract says, "Whoever signs this contract agrees to have a `display` method." In Listing 15-1, I declare a `Displayable` interface.

LISTING 15-1: Behold! An Interface!

```
package com.example.data;

public interface Displayable {
    void display();
}
```

Wait just a darn minute! The `display` method declaration in Listing 15-1 has a header but no body. No curly braces appear after `display()` — only a lonely-looking semicolon. What's going on here?

To answer the question, I'll let the code in Listing 15-1 speak for itself. If the code in the listing could talk, here's what the code would say:

> "As an interface, my `display` method has a header but no body. A class that claims to implement me (the `Displayable` interface) must provide (either directly or indirectly) a body for the `display` method. That is, a class that claims to implement `Displayable` must, in one way or another, provide its own code of the following kind:
>
> ```
> public void display() {
>     // Some statements go here
> }
> ```
>
> To implement me (the interface in Listing 15-1), the new code's `display` method must take no parameters and return nothing (also known as `void`)."

The `Displayable` interface is like a legal contract. The `Displayable` interface doesn't tell you what an implementing class already has. Instead, the `Displayable` interface tells you what an implementing class must declare in its own code.

In addition to displaying columns of numbers and tables, I may also want to summarize both kinds of data. How do you summarize a column of numbers? I don't know. Maybe you display the total of all the numbers. And how do you summarize a table? Maybe you display the table's column headings. How you summarize the data isn't my concern. All I care about is that you have some way to summarize the data.

So, I create code containing a second Java contract. The second contract says, "Whoever signs this contract agrees to have a `summarize` method." In Listing 15-2, I declare a `Summarizable` interface.

LISTING 15-2: Another Interface

```
package com.example.data;

public interface Summarizable {
    String summarize();
}
```

Any class claiming to implement the `Summarizable` interface must, by hook or by crook, provide an implementation of a `summarize` method — a method with no parameters that returns a `String` value.

REMEMBER

In the declaration of an interface, a particular method might have no body of its own. A method with no body is called an *abstract method.*

TECHNICAL STUFF

An interface's abstract method is automatically public whether you use the word `public` or not. For example, in Listing 15-2, the line

```
String summarize();
```

has the same meaning as

```
public String summarize();
```

If you enjoy typing the word `public`, feel free to do so. If not, leave that word out.

Implementing interfaces

Listing 15-3 implements the `Displayable` and `Summarizable` interfaces for a class named `ColumnOfNumbers`. To `display` a column of numbers, the listing prints all the numbers. To `summarize` a column of numbers, the listing prints the total of all the numbers.

LISTING 15-3: Implementing Two Interfaces

```
package com.example.data;

public class ColumnOfNumbers implements Displayable, Summarizable {
    private double numbers[];

    public ColumnOfNumbers(double[] numbers) {
        this.numbers = numbers;
    }

    @Override
    public void display() {
        for (double d : numbers) {
            System.out.println(d);
        }
    }

    @Override
    public String summarize() {
```

(continued)

LISTING 15-3: *(continued)*

```
        double total = 0.0;
        for (double d : numbers) {
            total += d;
        }
        return Double.toString(total);
    }
}
```

REMEMBER

When you implement an interface, you provide bodies for the interface's abstract methods.

Java's compiler is serious about the use of the `implements` keyword. If you remove either of the two method declarations from Listing 15-3 without removing the `implements` clause, you see some frightening error messages in your IDE's editor. Java expects you to honor the contract that the `implements` keyword implies. If you don't honor the contract, Java refuses to compile your code. So there!

TIP

You can use Java's error messages to your advantage. Start by typing some code containing the clause `implements Displayable, Summarizable`. Because of the `implements` clause, the editor displays an error mark and lists the names of the methods that you should have declared but didn't. In this section's example, those method names are `display` and `summarize`. After a few more mouse-clicks, the IDE generates simple `display` and `summarize` methods for you.

One final word about the code in Listing 15-3: The fact that the `display` method prints all the numbers and the `summarize` method prints the total is my choice alone, and it doesn't have to be your choice. If you want the `display` method to print "Hello, World!" and the `summarize` method to print today's news, that's up to you. The only demand that Java makes is as follows:

> Because the `ColumnOfNumbers` code begins with `implements Displayable, Summarizable`, you must tell Java what to do if and when someone calls this class's `display` and `summarize` methods.

Listing 15-4 contains another class that implements the `Displayable` and `Summarizable` interfaces. To `display` a table, the listing prints all the rows in the table. To `summarize` a table, the listing prints the table's header row.

LISTING 15-4: Another Class Implements the Interfaces

```
package com.example.data;

import module java.base;

public class Table implements Displayable, Summarizable {
    private Scanner diskFile;
    private ArrayList<String> lines = new ArrayList<>();

    public Table(String fileName) {
        try {
            diskFile = new Scanner(new File(fileName));
        } catch (FileNotFoundException e) {
            e.getMessage();
        }

        while (diskFile.hasNextLine()) {
            lines.add(diskFile.nextLine());
        }
    }

    @Override
    public void display() {
        for (String line : lines) {
            System.out.println(line);
        }
    }

    @Override
    public String summarize() {
        return lines.get(0);
    }
}
```

In Listings 15-3 and 15-4, notice several uses of the `@Override` annotation. In Chapter 10, you use `@Override` to signal the replacement of a method that's already been declared in a superclass. But you can also use `@Override` to signal an interface method's implementation. That's what I do in Listings 15-3 and 15-4.

Putting the pieces together

The code in Listing 15-5 makes use of all the stuff in Listings 15-1 to 15-4.

LISTING 15-5: Getting the Most Out of Your Interfaces

```
import com.example.data.*;

public class UseColumnAndTable {

    public static void main(String[] args) {
        double numbers[] = {21.7, 68.3, 5.5};
        var column = new ColumnOfNumbers(numbers);

        displayMe(column);
        summarizeMe(column);

        var table = new Table("MyTable.txt");

        displayMe(table);
        summarizeMe(table);
    }

    static void displayMe(Displayable displayable) {
        displayable.display();
        System.out.println();
    }

    static void summarizeMe(Summarizable summarizable) {
        System.out.println(summarizable.summarize());
        System.out.println();
    }
}
```

With the `MyTable.txt` file shown in Figure 15-1, the output from Listing 15-5 is shown in Figure 15-2.

```
Name  ID Balance
Barry 01 19.51
Carol 02 100.35
Myrna 03 10.07
```

FIGURE 15-1: The `MyTable.txt` file.

Feast your eyes on the `displayMe` method in Listing 15-5. What kind of parameter does the `displayMe` method take? Is it a `ColumnOfNumbers`? No. Is it a `Table`? No.

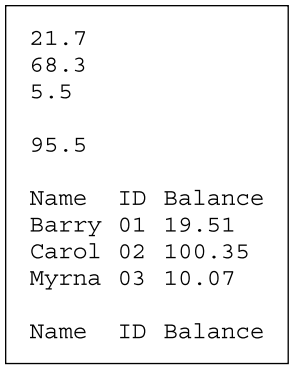


FIGURE 15-2: Running the code in Listing 15-5.

The `displayMe` method knows nothing about `ColumnOfNumbers` instances or `Table` instances. All the `displayMe` method knows about is things that implement `Displayable`. That's what the `displayMe` method's parameter list says. When you hand something that implements the `Displayable` interface to the `displayMe` method, the `displayMe` method knows what it can do. The `displayMe` method can call the parameter's `display` method, because that parameter object is guaranteed to have a `display` method.

The same kind of thing is true about the `summarizeMe` method in Listing 15-5. How do you know that you can call `summarizable.summarize()` inside the body of the `summarizeMe` method? You can make this call because `summarizable` has to have a `summarize()` method. The rules about Java interfaces guarantee it.

That's the real power behind Java's interfaces.

TWO KINDS OF METHODS

Inside an interface declaration, any method without a body is called an *abstract method.* You can also put methods with bodies inside an interface declaration. A method with a body is called a *default method*. In an interface's code, each default method declaration starts with the `default` keyword:

```
public interface MyInterface {

    void method1();

    default void method2() {
```

(continued)

(continued)

```
            System.out.println("Hello!");
        }
    }
```

In `MyInterface`, `method1` is an abstract method, and `method2` is a default method. If you create a class that implements `MyInterface`, like so:

```
class MyClass implements MyInterface
```

then your newly declared `MyClass` must declare its own `method1` and provide a body for `method1`. Optionally, your `MyClass` may declare its own `method2`. If `MyClass` doesn't declare its own `method2`, then `MyClass` inherits a `method2` body from `MyInterface`.

TRY IT OUT

In this section, the `ColumnOfNumbers` and `Table` classes implement the `Displayable` and `Summarizable` interfaces. What about a `Deletable` interface? Any class implementing the `Deletable` interface must have its own `delete` method.

CONTROL ALT DELETE

Create the `DeletableColumnOfNumbers` class — a subclass of the `ColumnOfNumbers` class. In addition to all the things `ColumnOfNumbers` does, the `DeletableColumnOfNumbers` class also implements the `Deletable` interface. When you `delete` a column of numbers, you set the values of each of its entries to `0.0`.

Create the `DeletableTable` class — a subclass of the `Table` class. In addition to all the things `Table` does, the `DeletableTable` class also implements the `Deletable` interface. When you `delete` a table, you remove all rows except the first (table heading) row. (***Hint:*** If you call the `lines` list's `remove` method starting from the `1` row and going to the `lines.size()` row, you won't be happy with the results. A call to the `remove` method modifies the list immediately, and that can mess up your loop.)

Abstract Classes

Consider the wondrous variety of life on planet earth:*

- A gelada monkey spends the day on a grassy plateau. But at night the gelada goes for a snooze on the rocky, perilous edge of a mountain cliff. With any luck, the sleeping monkey doesn't toss and turn much.
- A Pompeii worm lives in an underwater tube. The temperature by the worm's head is about 72 degrees Fahrenheit (22 degrees Celsius). But at the other end of the worm, the water temperature is normally 176 degrees Fahrenheit (80 degrees Celsius). If you know one of these worms personally, don't buy any warm socks for it.
- A sea squirt lives part of its life as an animal. At a certain point in its life cycle, the sea squirt attaches itself permanently to a rock and then digests its own brain, effectively turning itself into a plant.
- A tiny water bear can survive 12 days (and maybe more) with no atmosphere in the vacuum of outer space. Even the cosmic radiation in outer space doesn't harm a water bear. In 2017, a team of scientists concluded that water bears would survive most mass extinction events, including nearby supernova blasts and large asteroid impacts. So that's what I want to be in my next life — a water bear.

With so much biological diversity, there's very little I can say that applies to every animal on earth. But, I can certainly point to two properties: Every animal has a particular weight (measured in pounds, kilograms, or even picograms), and every animal makes (or, possibly, doesn't make) a characteristic sound. Listing 15-6 has the complete scoop.

---

* See `www.smithsonianmag.com/science-nature/ethiopias-exotic-monkeys-147893502`
`https://serc.carleton.edu/microbelife/topics/marinesymbiosis/pompeii.html`
`www.psychologytoday.com/blog/choke/201207/how-humans-learn-lessons-the-sea-squirt`
`www.esa.int/Our_Activities/Human_Spaceflight/Research/Tiny_animals_survive_exposure_to_space`
`www.nature.com/articles/s41598-017-05796-x`

LISTING 15-6: **What a Programmer Knows about Animals**

```
package com.example.species;

public class Animal {
    private double weight;
    private String sound;

    public Animal(double weight, String sound) {
        this.weight = weight;
        this.sound = sound;
    }

    public String getSound() {
        return sound;
    }

    public double getWeight() {
        return weight;
    }
}
```

While I was typing the code for the `Animal` class, I had to stop and correct several typing mistakes. The mistakes weren't really my fault: My cat was walking back and forth across my computer keyboard. And that brings me from the subject of all animals to the topic of house pets.

A house pet is an animal. But every house pet has a name — like Fluffy, Spot, or Princess. And every house pet has a recommended routine for taking care of the pet.

Of course, the care routines differ greatly from one kind of pet to another. If I had a dog, I'd have to walk the dog. But I'd never try to walk a cat. In fact, I don't even let our cat out of the house. So, when I define my `HousePet` class, I want to be vague about pet care instructions. And in Java, a class that's somewhat vague is called an *abstract class*. Listing 15-7 has an example.

LISTING 15-7: **What It Means to Be a House Pet**

```
package com.example.species;

import static java.lang.System.out;

public abstract class HousePet extends Animal {
    private String name;
```

```
    public HousePet(String name, double weight, String sound) {
        super(weight, sound);
        this.name = name;
    }

    public String getName() {
        return name;
    }

    abstract public void howToCareFor();

    public void about() {
        out.print(name + " weighs " + getWeight() + " pounds");
        out.print(getSound() != null ? (" and says '" + getSound() + "'") : "");
        out.println(".");
    }
}
```

On the first line of Listing 15-7, the keyword `abstract` tells Java that `HousePet` is an abstract class. Because `HousePet` is an abstract class, `HousePet` can have an abstract method. And in Listing 15-7, `howToCareFor` is an abstract method. An abstract method has a header but no body. In an abstract method's declaration, there are no curly braces — only a semicolon where curly braces would normally appear.

So, when you try to execute the `howToCareFor` method, what happens? Well, you can't really execute the `howToCareFor` method in Listing 15-7. In fact, you can't even create an instance of the abstract class declared in Listing 15-7. The following lines of code are illegal:

```
// VERY BAD CODE:
HousePet myPet = new HousePet("Boop", 12.0, "Meow");
var yourPet = new HousePet("Pawz", 22.5, "Woof");
```

An abstract class has no life of its own. To use an abstract class, you have to create an ordinary (non-abstract) class that extends the abstract class. In the ordinary class, all methods have bodies. So, everything works out.

CROSS REFERENCE

Before you walk away from Listing 15-7, notice the `super(weight, sound)` call in that listing. As in Chapter 10, the keyword `super` triggers a call to the superclass's constructor. In Listing 15-7, calling `super(weight, sound)` is like calling the `Animal(double weight, String sound)` constructor from Listing 15-6. The constructor assigns values to the new object's `weight` and `sound` fields.

Caring for your pet

Here's a quotation from the book *Java For Dummies*, 9th Edition:

> "To use an abstract class, you have to create an ordinary (non-abstract) class that extends the abstract class."

So, to use the `HousePet` class in Listing 15-7, you have to create a class that extends the `HousePet` class. The code in Listing 15-8 extends the abstract `HousePet` class and provides a body for the method named `howToCareFor`.

LISTING 15-8: It's a Dog's Life

```
package com.example.species;

import static java.lang.System.out;

public class Dog extends HousePet {
    private int walksPerDay;

    public Dog(String name, double weight, int walksPerDay) {
        super(name, weight, "Woof");
        this.walksPerDay = walksPerDay;
    }

    @Override
    public void howToCareFor() {
        out.print("Walk " + getName());
        out.println(" " + walksPerDay + " times each day.");
    }
}
```

In addition to having a `name`, a `weight`, and a `sound`, every dog gets walked a certain number of times per day. And now, because of the `howToCareFor` method's body, you know what caring for a dog means: It means walking the dog a certain number of times each day. It's a good thing the `howToCareFor` method is abstract in the `HousePet` class. You wouldn't necessarily want to walk some other kind of pet.

Take, for example, a domestic cat. "Caring" for a cat may mean not bothering it too often. And cats have other characteristics — characteristics that don't apply to dogs. For example, some cats go outdoors; others don't. You can make `walksPerDay` be 0 for an indoor cat, but that feels like cheating. Instead, each cat can have a `boolean` value representing the cat's indoor/outdoor status. Listing 15-9 has the code.

LISTING 15-9: How to Be a Cat

```
package com.example.species;

public class Cat extends HousePet {
    private boolean isOutdoor;

    public Cat(String name, double weight, boolean isOutdoor) {
        super(name, weight, "Meow");
        this.isOutdoor = isOutdoor;
    }

    @Override
    public void howToCareFor() {
        System.out.println (isOutdoor ?
               "Let " : "Do not let " + getName() + " outdoors.");
    }
}
```

Both the `Dog` and `Cat` classes are subclasses of the `HousePet` class. And, because of the abstract method declaration in Listing 15-7, both the `Dog` and `Cat` classes must have `howToCareFor` methods. But the `howToCareFor` methods in the two classes are quite different. One method refers to a `walksPerDay` field; the other method refers to an `isOutdoor` field. And because the `HousePet` class's `howToCareFor` method is abstract, there's no default behavior. Either the `Dog` and `Cat` classes implement their own `howToCareFor` methods or the `Dog` and `Cat` classes can't claim to extend `HousePet`.

TECHNICAL STUFF

This paragraph describes a picky detail, and you should ignore it if you have any inclination to do so: The `Dog` and `Cat` classes must implement the `howToCareFor` method because the `Dog` and `Cat` classes aren't abstract. If the `Dog` and `Cat` classes were abstract (that is, if they were abstract classes extending the abstract `HousePet` class), then the `Dog` and `Cat` classes would not have to implement the `howToCareFor` method. The `Dog` and `Cat` classes could pass the implementation buck to their own subclasses. For that matter, an abstract class that implements an interface doesn't have to provide bodies for all the interfaces' abstract methods. Abstract classes can take advantage of many little loopholes. But, to use these loopholes, you have to create some exotic programming examples. So, in this chapter, I simplify the story and write that (a) a class that extends an abstract class must provide bodies for the abstract class's abstract methods and (b) a class that implements an interface must provide bodies for the interface's abstract methods. It's not exactly true, but it's good enough for now.

If you live in a very small apartment, you may not have room for a dog or a cat. In that case, Listing 15-10 is for you.

LISTING 15-10: You May Grow Up to Be a Fish

```
package com.example.species;

public class Fish extends HousePet {

    public Fish(String name, double weight) {
        super(name, weight, null);
    }

    @Override
    public void howToCareFor() {
        System.out.println("Feed " + getName() + " daily.");
    }
}
```

I could go on and on creating subclasses of the `HousePet` class. Many years ago, our daughter had some pet mice. Caring for the mice meant keeping the cat away from them.

In Java, subclasses multiply like rabbits.

Using all your classes

Your work isn't finished until you've tested your code. Most programs require hours, days, and even months of testing. But, for this chapter's `HousePet` example, I do only one test. The test is in Listing 15-11.

LISTING 15-11: The Class Menagerie

```
import com.example.species.*;

void main() {
    var dog1 = new Dog("Fido", 54.7, 3);

    var dog2 = new Dog("Rover", 15.2, 2);

    var cat1 = new Cat("Felix", 10.0, false);

    var fish1 = new Fish("Bubbles", 0.1);

    dog1.howToCareFor();
    dog2.howToCareFor();
    cat1.howToCareFor();
    fish1.howToCareFor();
```

```
        dog1.about();
        dog2.about();
        cat1.about();
        fish1.about();
    }
```

When you run the code in Listing 15-11, you get the output shown in Figure 15-3.

```
Walk Fido 3 times each day.
Walk Rover 2 times each day.
Do not let Felix outdoors.
Feed Bubbles daily.
Fido weighs 54.7 pounds and says 'Woof'.
Rover weighs 15.2 pounds and says 'Woof'.
Felix weighs 10.0 pounds and says 'Meow'.
Bubbles weighs 0.1 pounds.
```

FIGURE 15-3: Please don't pet the Pompeii worm.

Notice how the code in Listing 15-11 seamlessly and effortlessly calls many versions of the `howToCareFor` method. With the `dog1.howToCareFor()` and `dog2.howToCareFor()` calls, Java executes the method in Listing 15-8. With the `cat1.howToCareFor()` call, Java executes the method in Listing 15-9. And, with the `fish1.howToCareFor()` call, Java executes the method in Listing 15-10 — it's like having a big `if` statement without writing the `if` statement's code. When you add a new class for a pet mouse, you don't have to enlarge an existing `if` statement. There's no `if` statement to enlarge.

Notice also how the `about` method in the abstract `HousePet` class keeps track of the object that called it. For example, when you call `dog1.about()` in Listing 15-11, the `HousePet` class's nonspecific `about` method knows that the sound `dog1` makes is `Woof`. Everything falls into place very nicely.

TRY IT OUT

Do you like abstract art? You can use abstract classes to create abstract art!

ASCII ART

Create an abstract class named `Shape`. The `Shape` class has a `size` field (of type `int`) and an abstract `show` method. Extend the abstract `Shape` class with two other classes: a `Square` class and a `Triangle` class. In the bodies of the `Square` and `Triangle` classes' `show` methods, place the code that creates a text-based rendering of the shape in question. For example, a `Square` of size 5 looks like this:

```
 _______
|       |
|       |
|       |
 _______
```

A `Triangle` of size 2 looks like this:

```
 /\
/  \
----
```

Relax! You're Not Seeing Double!

If you've read this chapter's earlier sections on interfaces and abstract methods, your head might be spinning. Both interfaces and abstract classes have abstract methods. But the abstract methods play slightly different roles in these two kinds of reference types. How can you keep it all straight in your mind?

The first thing to do is to remember that no one learns about object-oriented programming concepts without getting lots of practice in writing code. If you've read this chapter and you're confused, that may be a good thing. It means you've understood enough to know how complicated this stuff is. The more code you write, the more comfortable you'll become with classes, interfaces, and all these other ideas.

The next thing to do is to sort out the differences in the way you declare abstract methods. Table 15-1 has the story.

TABLE 15-1 Using (or Not Using) Abstract Methods

	In an Ordinary (Non-Abstract) Class	In an Interface	In an Abstract Class
Are abstract methods allowed?	No	Yes	Yes
Can a method declaration contain the `abstract` keyword?	No	Yes	Yes
Can a method declaration contain the `default` keyword (meaning "not abstract")?	No	Yes	No
With neither the `abstract` nor the `default` keyword, a method is:	Not abstract	Abstract	Not abstract

Both interfaces and abstract classes have abstract methods. So you may be wondering how you should choose between declaring an interface and declaring an abstract class. In fact, you might ask three professional programmers how interfaces and abstract classes differ from one another. If you do, you may get five different answers. (Yes, five answers; not three answers.)

Interfaces and abstract classes are similar beasts. But the basic idea is about the relationships among things.

- **Extending a subclass represents an *is a* relationship.**

  Think about the relationships in this chapter's earlier section "Abstract Classes." A house pet is an animal. A dog is a house pet. A cat is a house pet. A fish is a house pet.

- **Implementing an interface represents a *can do* relationship.**

  Think about the relationships in this chapter's earlier section "The Java Interface." The class declaration in Listing 15-3 says `implements Displayable, Summarizable`. With these words, the code promises that each `ColumnOfNumbers` object can be displayed and summarized. Later in the same listing, you make good on the promise by declaring a `display` method and a `summarize` method.

If you want more tangible evidence of the difference between an interface and an abstract class, consider this: A class can implement many interfaces, but a class can extend only one other class, even if that one class is an abstract class. So, after you've declared

```
public class Dog extends HousePet
```

you can't also make `Dog` extend a `Friend` class. But you can make `Dog` implement a `Befriendable` interface. And then you can make the same `Dog` class implement a `Trainable` interface. (By the way, I've tried making my `Cat` class implement a `Trainable` interface but, for some reason, it never works.)

And, if you want an even *more* tangible difference between an interface and an abstract class, I have one for you: An interface can't contain any nonstatic, nonfinal fields. For example, if the `HousePet` class in Listing 15-7 were an interface, it couldn't have a `name` field. That simply wouldn't be allowed.

So there. Interfaces and abstract classes are different from one another. But if you're new at the game, you shouldn't worry about the difference. Just read as much code as you can, and don't get scared when you see an abstract method. That's all there is to it.

IN THIS CHAPTER

- Displaying a window
- Using buttons and text fields
- Responding to mouse-clicks

Chapter 16
Oooey GUI Was a Worm

There's a wonderful old joke about a circus acrobat jumping over mice. Unfortunately, I'd get sued for copyright infringement if I included the joke in this book.

Anyway, the joke is about starting small and working your way up to bigger things. That's what you do when you read *Java For Dummies*, 9th Edition.

Most of the programs in this book are text-based. A *text-based* program has no windows, no dialog boxes — nothing of that kind. In a text-based program, the user types characters in a command window, and the program displays output in the same command window.

These days, very few publicly available programs are text-based. Almost all programs use a *GUI* — a *g*raphical *u*ser *i*nterface. So if you've read every word of this book so far, you're probably saying to yourself, "When am I going to find out how to create a GUI?"

Well, now's the time! This chapter introduces you to the world of GUI programming in Java.

Getting Started with GUI Applications

Java comes with three sets of classes for creating GUI applications:

- **The Abstract Window Toolkit (AWT):** The original set of classes, dating back to JDK 1.0.

  Classes in this set belong to packages whose names begin with `java.awt`. Components in this set have names like `Button`, `Frame`, `TextField`, and so on.

  The AWT implements only the kinds of components that were available on all common operating systems in the mid-1990s. So, using AWT, you can add a button to your application, but you can't easily add a table or a tree.

- **Java Swing:** A set of classes created to fix some of the difficulties posed by the use of the AWT. Swing was introduced with Java 1.1 in 1997.

  Classes in this set belong to packages whose names begin with `javax.swing`. Components in this set have names like `JButton`, `JTextField`, `JFrame`, and so on.

  A Swing program looks the same on every operating system. But Swing components aren't pretty. A `JButton` looks primitive and crude compared to a Mac button or a Windows button.

- **JavaFX:** The newest set of GUI classes in Oracle standard Java.

  Classes in this set belong to packages whose names begin with `javafx`.

  JavaFX supports over 60 kinds of components. (Sure, you want a `Button` component. But do you also want an `Accordion` component? JavaFX has one.) In addition, JavaFX supports multitouch operations and takes advantage of each processor's specialized graphics capabilities.

This chapter introduces JavaFX.

Setting up your computer

Chapter 2 tells you to create Java programs using any IDE that works for you. In most chapters, you can use Eclipse, VSCode, BlueJ, or even Notepad with the Windows MS-DOS prompt. But in this chapter, I guide you through the creation of a JavaFX app using IntelliJ IDEA.

Don't get me wrong. IntelliJ isn't the only IDE that lets you create JavaFX apps. But a typical JavaFX app consists of several folders, each with several files. You want an IDE to help you keep track of the moving parts. In this chapter, I have to pick one IDE to write about. So I pick IntelliJ.

TIP

Names can be confusing. The letters *IDE* stand for *Integrated Development Environment*. The most commonly used IDE for Java program development is *IntelliJ IDEA*, sometimes known simply as *IntelliJ* and occasionally known as *IDEA*. The company that makes IntelliJ IDEA used to be called *IntelliJ*. But in 2003, the company changed its name to *JetBrains*.

CROSS REFERENCE

If you don't have IntelliJ IDEA on your computer, you can find installation instructions in Appendix A. You can find that appendix by visiting `http://www.dummies.com/go/javafd9e`.

Installing the JavaFX plug-in

To make sure that IntelliJ can create a JavaFX app, you need IntelliJ's JavaFX plug-in. To get the plug-in, do the following:

1. **Go to IntelliJ's main menu. On Windows, choose File ➪ Settings ➪ Plugins. On a Mac, choose IntelliJ IDEA ➪ Settings ➪ Plugins.**

   After doing this, you may notice two tabs: one labeled Marketplace and another labeled Installed. (See Figure 16-1.)

2. **Select the Marketplace tab.**

   At the top of the Marketplace list, there's a search field. (Refer to Figure 16-1.)

3. **In the search field, type** JavaFX**.**

   As a result, IntelliJ shows you a list of relevant plug-ins. (Yup! Refer to Figure 16-1.) Each item in the list has an Install button or a selected check box.

   One of the list's items has the one-word name *JavaFX*. (At least, that's the plug-in's name in late 2024.) The item's fine print says *JetBrains s.r.o.*, which stands for JetBrains Společnost s ručením omezeným. If that item has a check box that's been selected, you already have the JavaFX plug-in.

4. **If the JavaFX item has an Install button, click that button.**

   When you do, IntelliJ installs its JavaFX plug-n.

TIP

If you can't decide which plug-in you want, install more than one of them. Having too many plug-ins is usually better than having no plug-ins.

TIP

The JetBrains company makes two versions of IntelliJ IDEA: the free Community Edition and the paid Ultimate Edition. If you can't install the JavaFX plug-in on the Community Edition, try the Ultimate Edition's 30-day free trial.

In addition to IntelliJ's JavaFX plug-in, you need an application named Scene Builder.

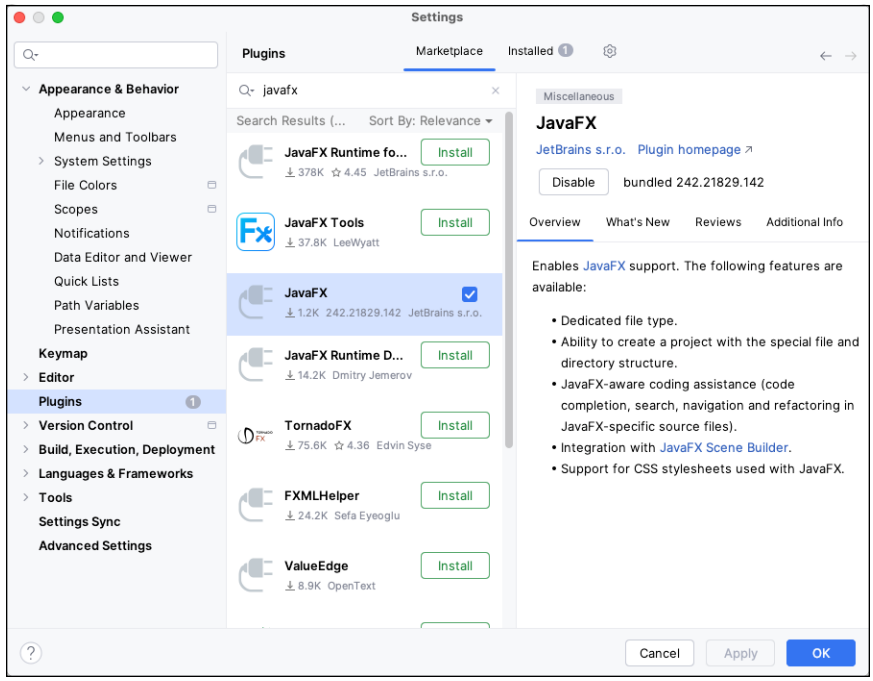


FIGURE 16-1: The Plugins option of IntelliJ's Settings dialog box.

Installing Scene Builder

Imagine you're at the Louvre, looking at the *Mona Lisa*. You don't think, "da Vinci added a face, and then he put a smile on the face, and then he added a body and then a background." The painting doesn't progress from one action to another. Instead, the painting simply *is*.

In the same way, you don't have to add text fields and buttons, one by one, to a window in a GUI application. Instead, you can simply describe a window's elements. You can say, "Here's how the window should look when it's all finished being rendered." Then you can let Java decide what steps to take to display the window's elements.

So, how do you describe an application window's elements? You can do what programmers usually do — type a bunch of text in a code editor. But there's a better way. The most natural way for you to describe visual elements is to "draw" them.

What if there was a way for you to draw an app's window and have the computer turn your drawing into code that Java can interpret?

Yes! There's a way to do this. Using an app called Scene Builder, you draw a window's components with your mouse. Scene Builder automatically turns your visual description into code.

Installing Scene Builder is like installing most software. Here's how you do it:

1. **Visit** `http://gluonhq.com/products/scene-builder`**.**
2. **Click the Download button.**

   When you do, a list of download options appears.
3. **Click the button corresponding to your computer's operating system (Windows, Mac, or Linux).**

   As a result, the download begins. On a Windows computer, you get an `.msi` file. Double-click the file to begin the installation.

   On a Mac, you get a `.dmg` file. Depending on your Mac web browser's setting, the browser may or may not expand the `.dmg` file automatically. If not, double-click the `.dmg` file to begin the installation.
4. **Follow the installation routine's instructions.**

   On a Windows computer, you accept a bunch of defaults. One of those defaults specifies the location on your hard drive where Scene Builder is to be installed. (On my Windows computer, that location is `C:\Users\bburd\AppData\Local\SceneBuilder`. Don't ask me why.) You'll need to know this location, so be sure to make a note of it.

   On a Mac, you drag the Scene Builder's icon to the `/Applications` folder.

That's it! You've installed Scene Builder.

Creating a bare-bones JavaFX project

After installing the JavaFX plug-in and Scene Builder, you can create a GUI application with clicks and keystrokes. Here's what you do:

1. **Follow the instructions in the earlier section "Setting up your computer."**
2. **From IntelliJ's menu bar, choose File ➪ New ➪ Project.**

   The New Project dialog box appears. There's a list of items along the left side of the dialog box. (See Figure 16-2.)
3. **In the dialog box's list, select JavaFX. (Refer to Figure 16-2.)**
4. **In the main body of the New Project dialog box, type a name for your project.**

   Typically, you don't have to mess with any other elements in the New Project dialog box's body.

WARNING

Sitting here on my temporal perch in 2024, I notice a glitch in running JavaFX programs on a Mac. When I have some version of Java 23 in the dialog box's JDK drop-down menu, my JavaFX programs crash with a cryptic `NSInternalInconsistencyException` message. Fortunately, when I select some version of Java 17, the programs run just fine.

5. **At the bottom of the dialog box, click Next.**

   As a result, another New Project dialog box appears. This box offers to add additional libraries to your project. For this chapter's example, you don't need any of these libraries. Leave their check boxes unselected.

6. **At the bottom of the dialog box, click Create.**

   IntelliJ may ask you if you want your project in the current window or in a new window. Pick whichever you want.

   One way or another, IntelliJ shows you a window with your brand-new JavaFX project. (See Figure 16-3.)

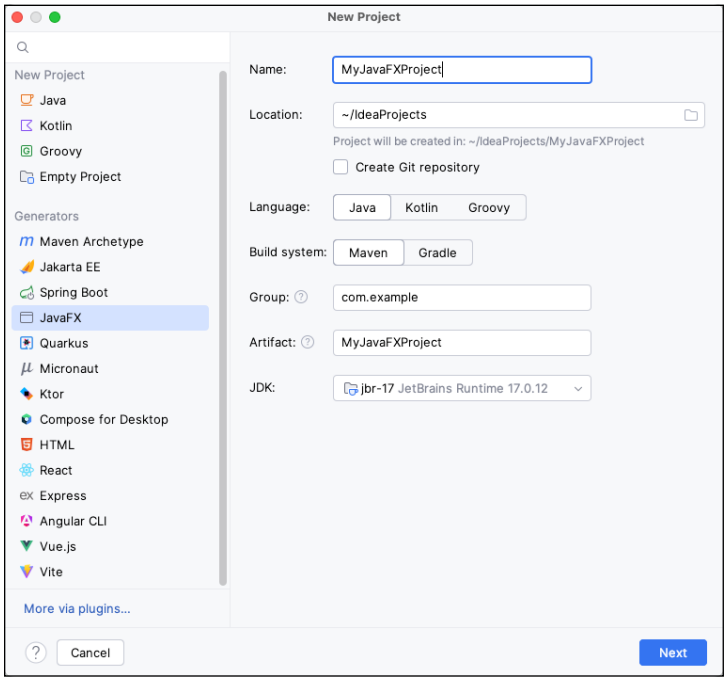


FIGURE 16-2: The New Project dialog box.

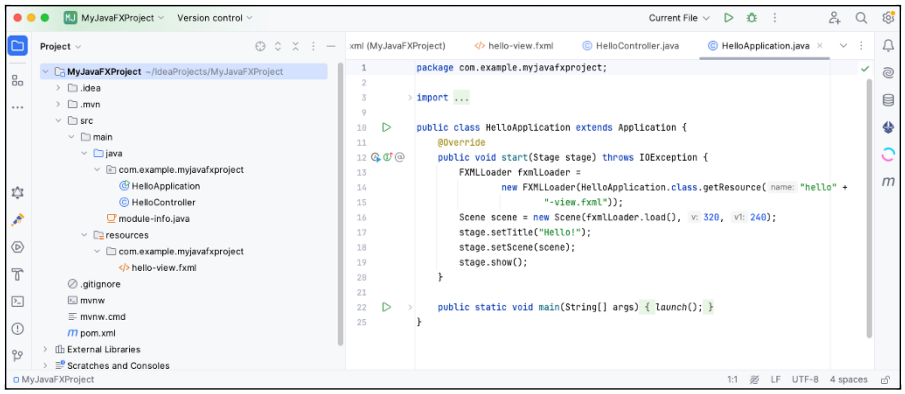


FIGURE 16-3: A new JavaFX project.

YOUR SCREEN DOESN'T LOOK LIKE FIGURE 16-3?

If you don't see the stuff in Figure 16-3, don't panic. Among all the reasons this can happen, these are the two most common:

- **You missed one of the steps in this chapter's instructions.**

  I've made mistakes like that millions of times. If you can't figure out what you missed, send me an email. The address is `JavaForDummies@allmycode.com`.

- **The IntelliJ JavaFX plug-in has changed the way it behaves.**

  Figure 16-3 is what I see as I write this chapter in 2024. I have no doubt that Hermes, the Greek god of authors, will play tricks on me between 2024 and the time you read this book. Hermes will command the IntelliJ company to change its JavaFX plug-in so that the window you see isn't what I've pasted into 16-3. If this is the case, check the code that you can download from `https://javafordummies.allmycode.com`. I've included a copy of this chapter's JavaFX project in that download. You can copy parts of my project or open my entire project with IntelliJ. And, as always, it never hurts to send me an email.

Running your bare-bones JavaFX project

In the previous section, you use IntelliJ IDEA to create a JavaFX project. Here's a brief overview of three files in the JavaFX project:

- `HelloApplication.java`: This file contains the two methods that get the ball rolling. One of them is the omnipresent `main` method. The other is a method named `start`.

  The `start` method sets up your JavaFX app's initial scaffolding. In particular:

  - This method calls the `Scene` constructor. The *scene* is a list of all the items that appear at any moment on the user's screen. For example, the scene may include a column with a text field and a button.
  - JavaFX puts a new window, called a *stage*, on the user's screen. The `start` method connects the scene with this new window. Then the statement `stage.show()` makes this window appear.

- `HelloController.java`: This file defines actions that can take place while the JavaFX application runs. For example, the code

```
private Label welcomeText;

void onHelloButtonClick() {
    welcomeText.setText("Welcome to JavaFX Application!");
}
```

  says "When the user clicks the *Hello* button, display *Welcome to JavaFX Application!* on the `welcomeText` label.

- `hello-view.fxml`: This file describes the layout of the buttons, labels, and other items in your JavaFX application window. The file also connects the graphical components with actions in the aforementioned `HelloController.java` file. For more information, see this chapter's nearby sidebar "Code soup: mixing XML with Java."

To run all this code, right-click anywhere inside the `HelloApplication.java` file's Editor panel. Then select Run 'HelloApplication.main()'. After a brief pause, you'll see the stuff in Figure 16-4. It's a window with a button inside it. When you click the button, a Welcome message appears.

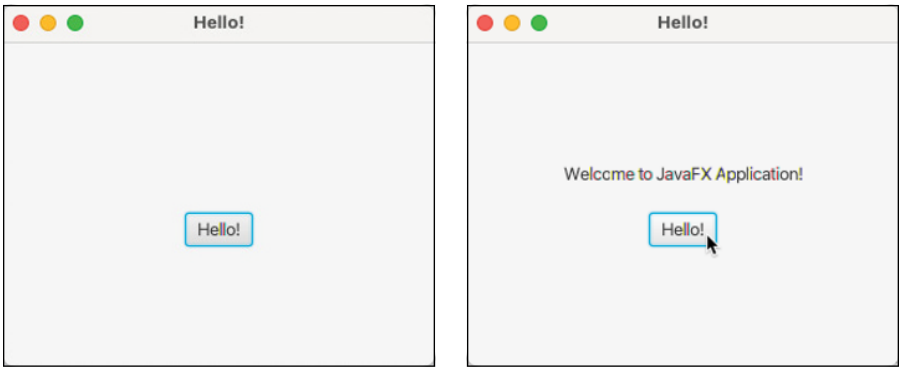


FIGURE 16-4: Running the sample JavaFX application.

CODE SOUP: MIXING XML WITH JAVA

How would a Java program describe the elements on the left side of Figure 16-4? The code may look something like this:

```
// This isn't real code.
var scene = new Scene();
var label = new Label();
scene.add(label);

var button = new Button();
button.setText("Hello!");
scene.add(label);

scene.show();
```

This "do this, then do that" approach is called *imperative programming*.

But doing this and then doing that isn't the best way to describe a window on a computer screen. With JavaFX, you don't describe a window by saying *how to build* the window. Instead, you describe the window by saying what's *already on* the window. This approach is called *declarative programming*.

Notice the declarative nature of this abridged version of IntelliJ's `hello-view.fxml` file:

```
<VBox alignment="CENTER" spacing="20.0">
    <padding>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0"/>
    </padding>

    <Label/>
```

(continued)

(continued)

```
        <Button text="Hello!"/>
    </VBox>
```

In this code, `VBox` indicates a column (a vertical box), `Label` is a place to display some words, and `Button` is (you guessed it) a button. This code *declares* the elements in the app's user interface. It does this without the clutter that comes from *commanding* Java to construct the interface.

The code in `hello-view.fxml` has XML (*eXtensible Markup Language*) tags. If you're familiar with HTML (the language of web pages), you might recognize some of the tricks in the XML code. If not, don't worry. Using a tool named Scene Builder, your computer writes the XML code on your behalf. To see what I mean, keep reading.

Adding Stuff to Your JavaFX Project

GUI programs involve visual elements, and the best way to describe visual elements is to draw them. You can draw them with a pencil or draw them with a mouse.

Using Scene Builder and a mouse, you can draw your app's layout and describe the app's components. Scene Builder automatically turns your drawing into XML code.

CROSS REFERENCE

You probably aren't surprised to hear that instructions for installing Scene Builder are in this chapter's "Installing Scene Builder" section. I just figure I'll point that out.

Where's my Scene Builder application?

IntelliJ IDEA isn't very good at finding Scene Builder on your computer. So, before you try to run Scene Builder, follow these steps:

1. **Go to IntelliJ's main menu. On Windows, choose File and, on a Mac, choose IntelliJ Idea. In the resulting submenu, choose Settings ➪ Languages & Frameworks ➪ JavaFX.**

   Doing so gets you what you see in Figure 16-5.

   In the main body of IntelliJ's Settings dialog box, you see a text field labeled Path to SceneBuilder.

2. **Click the Folder icon on the rightmost edge of the text field and navigate to the SceneBuilder app on your computer's hard drive.**

On my Mac, the text field says `/Applications/SceneBuilder.app`. (Refer to Figure 16-5.) On my Windows computer, the text field says `C:\Users\`*`my_user_name`*`\AppData\Local\SceneBuilder\SceneBuilder.exe`. (Refer to this chapter's "Installing Scene Builder" section.)

3. **Dismiss the Settings dialog box by clicking OK.**

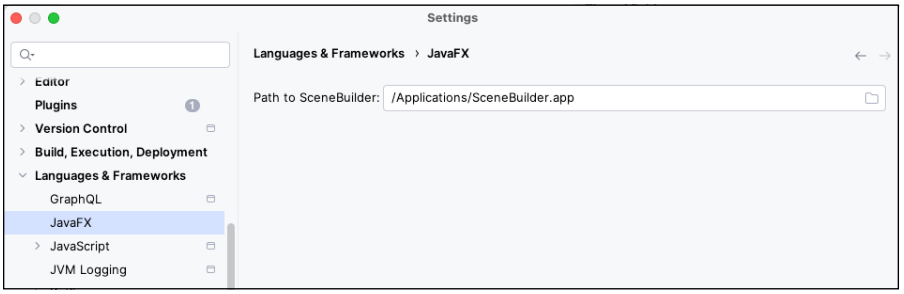


FIGURE 16-5: Hope and Crosby's "Road to Scene Builder."

Building a scene

Having told IntelliJ where to find Scene Builder, you're ready to add features to the JavaFX app. Here's how you do it:

1. **Find IntelliJ's Project tool window.**

   The Project tool window contains a tree of folders and files. (If you don't see IntelliJ's Project tool window, press the Project tool button. Appendix A, safely stored at `www.dummies.com/go/javafd9e`, has the details.)

2. **Expand the branches of the Project tool window's tree until you find the `hello-view.fxml` file.**

   The `hello-view.fxml` file is on the `src/main/resources/`*`your_project_package_name`* branch of the tree.

3. **Right-click the `hello-view.fxml` file. From the resulting context menu, choose Open in SceneBuilder.**

   If all goes well, a Scene Builder window appears on your screen. (If all doesn't go well, return to the "Where's my Scene Builder application?" section.

   The Scene Builder window contains several areas. (See Figure 16-6.)

4. **Along the left side of the Scene Builder window, look for the TextField component. You'll find it in the Controls part of the Library panel. (See Figure 16-7.)**

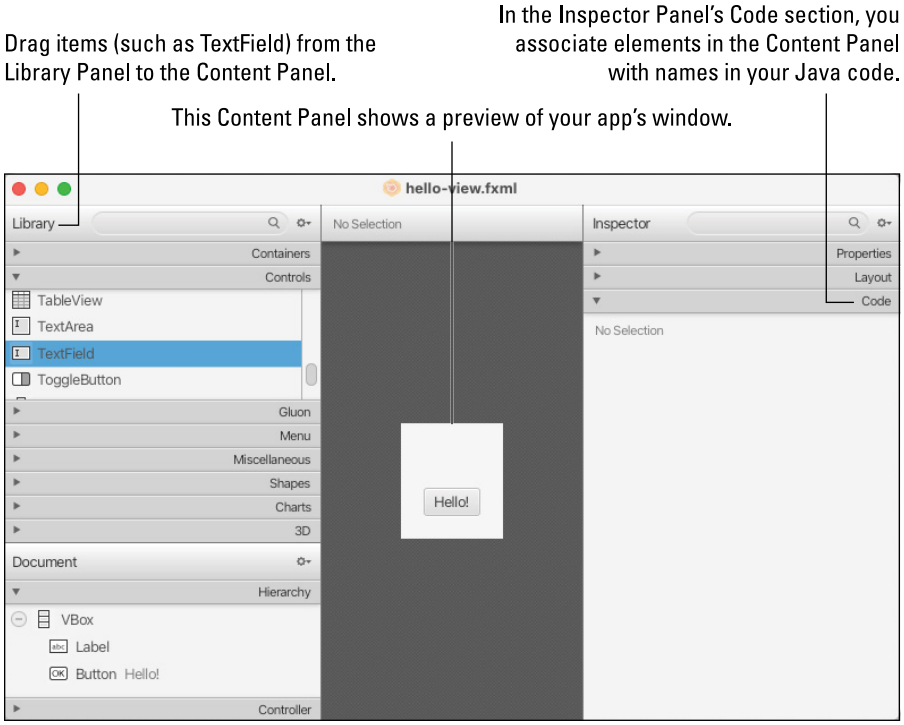


FIGURE 16-6: Scene Builder starts running.

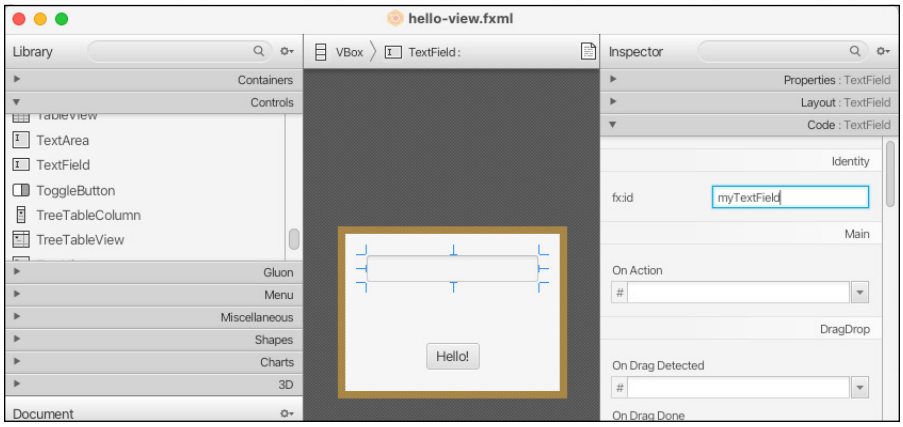


FIGURE 16-7: Your GUI app has a text field.

5. Using the mouse, click to select the TextField component and then drag it to the Content panel. In particular, drag it to the empty space above the *Hello!* button.

When you release the mouse, a text field appears in the Content panel above the *Hello!* button. Refer to Figure 16-7.)

6. **Click the mouse on the text field in the Content panel.**

   When you do, the Scene Builder's Inspector panel lights up with a set of features belonging to the TextField.

7. **In the Code part of the Inspector panel, look for the box labeled *fx:id*. (Refer to Figure 16-7.)**

8. **In that *fx:id* box, type the name** myTextField**.**

   Full disclosure: When you type something in the *fx:id* box, you're simply making up a variable name. Rather than type `textField`, you could type `montrose` or `hungadungaAndMcCormick`. But the name `myTextField` is more informative than those other two choices.

9. **Back in the Content panel, double-click the *Hello!* button.**

   When you do, the word *Hello!* on the face of the button becomes highlighted. Anything you type will replace that word *Hello!*

10. **On the face of the button, type** Capitalize**. (See Figure 16-8.)**

11. **With the Content panel's button still selected, look at the Code part of the Inspector panel.**

    Notice the Code part's OnAction value. (Refer to Figure 16-8.) The default is `onHelloButtonClick`, and there's no reason for you to change it. This tells JavaFX to call a method named `onHelloButtonClick` when the user clicks that button.

12. **From the Scene Builder's main menu, choose File ➪ Save.**

    I often forget to perform this step. Don't be like me. Don't forget this step!

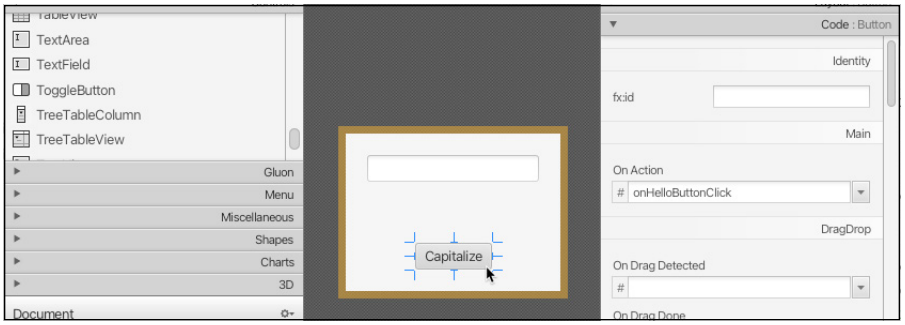


FIGURE 16-8: Tweaking the button.

Tweaking some code

In the previous section, you finish working with Scene Builder. Now return to the IntelliJ window and glance at the `hello-view.fxml` code in IntelliJ's Editor. I've copied the code right here and used boldface font for all the text that Scene Builder generated.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.VBox?>

<VBox alignment="CENTER" spacing="20.0" xmlns:fx="http://javafx.com/fxml/1"
   xmlns="http://javafx.com/javafx/22" fx:controller="com.example.
   myjavafxproject.HelloController">
    <padding>
        <Insets bottom="20.0" left="20.0" right="20.0" top="20.0"/>
    </padding>
    <TextField fx:id="myTextField" />

    <Label fx:id="welcomeText"/>
    <Button onAction="#onHelloButtonClick" text="Capitalize"/>
</VBox>
```

TIP

If the code in the Editor doesn't already have a `TextField` and a `Button` with the word `Capitalize`, you may need to update the contents of the Editor. To do so, navigate away from the `hello-view.fxml` file and then back to that file.

You still have to add some code to the project's `HelloController.java` file.

1. **Double-click on the `HelloController` branch in IntelliJ's Project tool window.**

   More specifically, look for the tree's `src/main/java/`*`your_project_package_name`*`/HelloController` branch. When you double-click this branch, the `HelloController.java` code appears in IntelliJ's Editor.

2. **Modify the `HelloController.java` file as shown in Listing 16-1.**

   IntelliJ has already created most of the code in Listing 16-1 for you. The only text you have to think about is the text I've highlighted with boldface font. Here's a quick tour of those parts of the code:

- You need `import javafx.scene.control.TextField` because you've added a `TextField` to your app. You may not have to type this declaration. IntelliJ may have added it automatically.
- The `myTextField` declaration (and the `@FXML` annotation immediately above it) tell Java to look in the stuff you created with Scene Builder for a `TextField` component whose *fx:id* is `myTextField`.

  For the scuttlebutt on Java annotations, refer to Chapter 10.

CROSS REFERENCE

- The `myTextField.getText().toUpperCase()` code tells Java to get whatever the user has typed in `myTextField` and display an all-uppercase version of that text in the `welcomeText` label.

LISTING 16-1: The Finished HelloController.java File

```
package com.example.fxdemo;

import javafx.fxml.FXML;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;

public class HelloController {
    @FXML
    private Label welcomeText;
    @FXML
    private TextField myTextField;

    @FXML
    protected void onHelloButtonClick() {
        welcomeText.setText(myTextField.getText().toUpperCase());
    }
}
```

Running your new project

Now you can put this chapter's hard work to the test:

1. **In the Project tool window, right-click the `HelloApplication` branch and choose Run 'HelloApplication.main()'.**

   Lo and behold! A new application window appears on your screen!

2. **Type something in the text field.**

   (Try to make it something nice.)

3. **Click the app's Capitalize button.**

   Wow! An all-uppercase version of the text field's code appears in your app's window. (See Figure 16-9.)

TRY IT OUT

Tweak the program in this section for some exciting results. (Okay. The results may be uninteresting to some people, but they're exciting to me!)

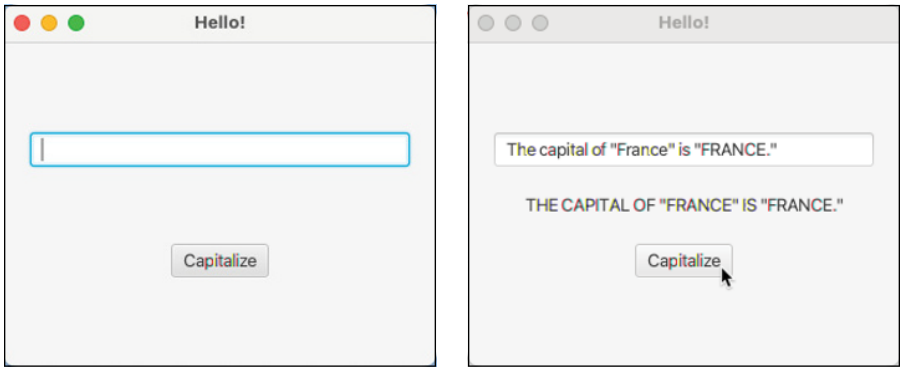


FIGURE 16-9: . . . But the *capitol* of France is Paris!

COPY TEXT TO A TEXT FIELD

Create a window with a button and two text fields. When the user clicks the button, Java copies text from the first text field to the second text field.

COPY BETWEEN TEXT FIELDS

Create a window with two buttons and two text fields. When the user clicks the first button, Java copies text from the first text field to the second text field. When the user clicks the second button, Java copies text from the second text field to the first text field.

CREATE A SMALL CALCULATOR

Create a window with two text fields, a button, and a label. The user types a number in one of the text fields and another number in the other text field. When the user clicks the button, Java displays the sum of the two numbers in the label.

Remember that Java distinguishes between `String` values like `"86"` and `int` values like `86`. Java reads `String` values from the text fields. But, to get the sum, you add two `int` values. Then you put a `String` value on the app's label.

5 The Part of Tens

IN THIS PART . . .

Explore the best resources for Java on the web.

Read the Dear Barry advice column.

IN THIS CHAPTER

- Checking out this book's website
- Finding resources from Oracle
- Reading more about Java

Chapter 17
Ten Packs of Java Websites

Before starting this chapter, I paused to count the number of tabs I had open in my web browser. I counted 64 tabs. What about you? Are you a tabaholic too?

This Book's Website

For all matters related to the technical content of this book, visit `https://javafordummies.allmycode.com`. And don't forget: If you have questions about anything you read in this book, send email to me at `JavaForDummies@allmycode.com`.

For Business Issues Related to This Book

For example, to ask, "How can I purchase 100 more copies of *Java For Dummies*?" visit `www.dummies.com`.

Download the Java Development Kit

For Oracle's official version of Java, visit `www.oracle.com/java/technologies`.

Your Grandparents' Java Download Site

People who want to run Java programs but don't *have* to write Java programs should visit `www.java.com`.

The Horse's Mouth

Check the official Java API documentation at `https://docs.oracle.com/en/java/javase/23/docs/api`. And, to settle any argument about the way the language behaves, read the rigorous Java language specification at `https://docs.oracle.com/javase/specs`.

Join Java User Groups

I happen to share leadership roles in the New York JavaSIG (`www.javasig.com`) and the Garden State Java User Group (`www.gsjug.org`). Both groups have regular online meetings and in-person meetings. Come join us!

Find the Latest News about Java

Bookmark these three sites:

- `https://foojay.io`
- `https://dev.java`
- `https://forums.oracle.com/ords/apexds/domain/dev-community`

Find News, Reviews, and Sample Code

For articles by the experts, visit InfoQ at `www.infoq.com` and TheServerSide at `www.theserverside.com`. You can always find good reading at these two sites.

Got a Technical Question about Anything?

If you're stuck and need help, search for answers and post questions at Stack Overflow (`https://stackoverflow.com`).

You can also post questions at Coderanch: "A friendly place for programming greenhorns" (`https://coderanch.com`). For questions specific to Java, visit `https://coderanch.com/c/java`.

Become Involved in the Future of Java

The *Java Community Process* (JCP) ". . . is the mechanism for developing standard technical specifications for Java technology. Anyone can register . . . and participate in reviewing and providing feedback for the Java Specification Requests (JSRs)." For more info about JCP membership, visit `www.jcp.org/en/participation/membership_ind`.

IN THIS CHAPTER

- What to learn
- How to learn more
- What to do when you're looking for a job
- Other free advice

Chapter 18

Ten Bits of Advice for New Software Developers

I enjoy hearing from the people who read my books. "Nice job!" one reader says. Another reader asks, "Can I run your book's examples on older versions of Java?" Yet another posts this comment: "You're Barry Burd. Does that mean you're related to Larry Bird?"

In all the questions I receive from readers, one popular theme is "What to do next?" More specifically, people ask me what else to learn, what else to read, how to get practice writing software, how to find work, and other questions of that kind. I'm flattered to be asked, but I'm reluctant to think of myself as an authority on such matters. No two people give you the same answers to these questions, and if you ask enough people, you're sure to find disagreement.

This chapter contains ten pieces of advice based on questions I've received from readers. But remember that, in addition to these ten hints for living and learning, I have one additional, overriding piece of advice:

> Think critically about the advice you receive. When in doubt, trust your intuition.

Collect opinions. Talk to people about the issues. Try things and, if they work (or even if they don't work but they show some promise), keep doing them. If they show no promise, try other things. Sharing is important, too. Don't forget to share.

How Long Does It Take to Learn Java?

The answer depends on you — on your goals, on your existing knowledge, on your capacity to think logically, on the amount of spare time you have, and on your interest in the subject.

The more excited you are about computer programming, the quicker you learn. The more ambitious your goals, the longer it takes to achieve them.

REMEMBER

There's no such thing as "knowing all about Java." No matter how much you know, you always have more to learn. I've written several Java books and, as far as I'm concerned, I've barely scratched the surface.

Which of Your Books Should I Read?

Funny you should ask! I've written several books, including this book *(Java For Dummies)* and another, similarly titled book, *Beginning Programming with Java For Dummies*.

Both books start from scratch, so you don't need to know anything about app development to read either of these books. But *Java For Dummies* goes twice as fast and covers twice as much material as *Beginning Programming with Java For Dummies*. Which book you read depends on your level of comfort with technical subjects. If you're in doubt about where to start, find some sample pages from any of these books to help you determine which book is best for you.

Are Books Other than Yours Good for Learning Java?

Jeanne Boyarsky and Victor Grazi have a book titled *Real-World Java*. The book has many good things going for it:

- It's published by Wiley.
- It covers the kinds of things all developers (especially Java developers) should know.
- I'm the book's technical proofreader.
- It includes a reference to my book.
- Jeanne and Victor are friends of mine.

So what are you waiting for? Buy *Real-World Java*!

Which Computer Programming Language(s) Should I Learn?

The answer depends on your goals. If you plan to write code for a living, the answer depends on the job opportunities where you live. The TIOBE Programming Community Index (`www.tiobe.com/tiobe-index`) provides monthly ratings for popular programming languages. But the TIOBE Index might not apply specifically to your situation. In June 2021, the Haskell language ranked only 47th among the languages used around the world. But maybe there's a hotbed of Haskell programming in the town where you live.

Do you want to write applications for large enterprises? Then Java is a must-have language. Do you want to do data science? If so, you should learn Python. Do you want to create web pages? Learn HTML, CSS, and JavaScript.

Which Skills Other than Computer Coding Should I Learn?

Sorry to disappoint you, but you're asking someone who has an axe to grind. I'm a college professor. I believe that no learning, no matter how impractical it might seem to be, is ever wasted.

If you insist on a more definitive answer, go learn a little about databases. Database work isn't necessarily coding, but it's important stuff. Also, read as much as you can about *software engineering* — the study of techniques for the effective design and maintenance of computer code. Don't be afraid of math, either (because

learning math stretches your logical-thinking muscles). And, whenever you can, hone your communication skills. The better you communicate, the more valuable your work is to other people.

How Should I Continue My Learning as a Software Developer?

Practice, practice, practice. Take the examples you find in my book (or anywhere else) and think of ways to change the code. Add an option here or a button there.

Find out what happens when you try to improve the code. If it works, think of another way to make a change. If it doesn't work, search the documentation for a solution to your problem. If the documentation doesn't help (and often, documentation doesn't help), search the web for answers to your problem. Post questions at an online forum. If you don't find an answer, put aside the problem for a while and let it incubate in your mind.

REMEMBER

You don't learn programming by only reading about it — you have to scrape some knuckles while writing code and seeking solutions. Only after trying, failing, and trying again can you appreciate the work involved in developing computer software.

How Else Should I Continue My Learning as a Developer?

How did you know that I have a second suggestion? I recommend finding like-minded people where you live and getting together with them regularly. These days, you can find tech user groups in almost every corner of the globe. Find a Java user group that meets in your area and attend the group's meetings frequently. If you're a novice, you might not understand much of the discussion, but you'll be exposed to the issues that concern today's Java developers.

Look for more tech groups and attend their meetings. Find meetings about other programming languages, other technologies, and other topics that aren't solely about technology. Meet people face-to-face and find out which topics will be in next year's books.

To complement those face-to-face meetings, search the web for screen-to-screen meetings. You can find free online technical sessions almost any day of the year.

How Can I Land a Job Developing Software?

Do all the things you'd normally do when you look for a job, but don't forget my answer to the previous question. User groups are fantastic places for networking.

Go to meetings and be a good listener. Don't think about selling yourself. Be patient and enjoy the ride. I landed a great consulting opportunity only after several years of attending one group's meetings. In the meantime, I learned a lot about software (and quite a bit about dealing with other people).

I Still Don't Know What to Do with My Life

That's not a question. But it's okay anyway.

Everyone has to make ends meet. If you manage to put food on your table, the next step is to find out what you love to do. I've spent a lifetime teaching college students, writing books, and developing computer code — and I love doing all of it. (Well, I love most of it. I detest grading papers, and I dislike proofreading my own work.)

Fortunately, I can make money teaching, writing, and developing. I could make more money working 9-to-5 for a big company, managing a software team, or creating the next big start-up, but I don't like doing those things. My life has been enriched because I do what I like doing, whether I'm working or not.

My advice is, find the best match of the things you like to do and the things that help you earn a living. Compromise if you must, but be honest with yourself about the things that make you happy. (Of course, these things shouldn't make other people unhappy.)

Finally, be specific about your likes and dislikes. For example, saying, "I'd like to be rich" isn't specific at all. Saying, "I'd like to create a great game" is more specific, but you can do better. Saying, "I like to design game software, but I need a partner who can do the marketing for me" is quite specific and makes quite a tidy set of goals.

If I Have Other Questions, How Can I Contact You?

That's easy. Send email to `javafordummies@allmycode.com`. Visit my ugly-but-informative website at `https://users.drew.edu/bburd/` and look for the page's Chat with Barry panel. Attach two tin cans to a very long string. Put a note in an old pneumatic tube. Train a carrier pigeon to fly to my office. Hire a chimpanzee to . . .

Index

Symbols

* (asterisk), 81, 287
&& (and operator), 112, 121
\ (backslash), 245
{} (curly braces), 46–47, 103, 104, 151
\\ (double backslash), 245
== (double equal signs), 102–103, 107, 108
++ (double plus signs), 83–88
""" (double quotes), 111
! (exclamation point), 144
> (greater than operator), 107
>= (greater than or equal to operator), 107
< (less than operator), 107
<= (less than or equal to operator), 107
! (not operator), 112
!= (not equal to operator), 107, 108
|| (or operator), 112, 116–117
/ (slash) operator, 81
_ (underscores), 61, 66

A

abstract classes
  abstract methods in, 424–425
  creating art, 423–424
  extending, 420–422
  general discussion, 417–419
  testing, 422–423
`abstract` keyword, 419
abstract methods, 411, 415, 419, 424–425
Abstract Window Toolkit (AWT), 428
access
  access modifiers for classes, 293–294
  default, 185–186, 298
  experiments with, 299–301
  for members in different classes, 295–299
  for members in same class, 295
  method-local variables, 294–295
  for modules, 292–293
  overview, 291–292
  packages
    formatting in, 289–291
    instances, 288–289
    naming, 282–284
    overview, 282
    subpackages, 287
    using `Player` class, 284–287
  private, 183–184, 296
  protected, 298–299
  public, 184, 297–298
  `static` fields and methods
    compact constructors, 303–304
    experiments with, 310–312
    general discussion, 301–303
    `import static` declaration, 306–307
    `main` methods, 308–310
access modifiers
  for classes, 293–294
  for members in classes, 295–299
  overview, 291, 292
accessor methods
  enforcing rules with, 185–186
  general discussion, 180–183
  making fields inaccessible, 183–185
  `Temperature` class, 208–211
`Account` class
  accessor methods, 180–186
  calculating interest, 171–173
  declaring variables and creating objects, 160–161
  `display` method, 164–171, 186
  displaying accounts, 164–171
  fields in, 158
  `getInterest` method, 173–177, 186
  passing values to methods, 174–175
  returning values from, 175–177
  using object fields, 161–162
accumulators, 144
`add` methods, 317
advice for software developers, 449–454

AI tools, 9
AirplaneFlight class, 219–220, 267
AllWords.txt file, 315–317
"Al's All Wet" song
  for statements, 142–143
  switch statements, 123–127
amountInAccount variable, 60, 61
and operator (&&), 112, 121
anEmployee variable, 238
Animal class, 418
annotations, 263–264
Apartment class, 261, 266
API (application programming interface), 36–37, 119, 381, 446
API identifiers, 37–38
applications, GUI
  adding visual elements, 436–442
  classes for creating, 428
  creating, 431–433
  installing JavaFX plugin, 429
  installing Scene Builder, 430–431
  running, 434–435
  setting up computer, 428–429
args array, 371–375
arguments, program, 371–375
ArithmeticException, 392, 393
array initializers, 353–355
ArrayList class
  add methods, 317
  arrays versus, 356–357
  autoboxing, 321–323
  enhanced for loop, 318–319, 324
  experiments with, 329
  general discussion, 314–317
  getting streams from, 336
  hasNext method, 323–324
  importing module, 319–320
  iterators, 323–324
  next method, 324
  overview, 328
  using generics, 320–321
arrays
  array initializers, 353–355
  ArrayList versus, 356–357
  checking for vacancies, 357–359
  components of, 350
  conditional operator, 370–371
  connecting to files, 360–361
  enhanced for loops, 353–355
  escape sequences, 352–353
  experiments with, 355–356, 364–365
  filling with values, 351–352
  general discussion, 349–351
  inserting values into, 361–362
  length of, 359
  of objects, 365–371
  program arguments, 371–375
  Room class, 367–370
  sort method, 363–364
  tab stops, 352–353
  writing to files, 359–360
Arrays class, 364
Arrays.toString method, 362
ASCII character encoding, 77
assignment operators, 88–90
assignment statements, 62
asterisk (*), 81, 287
atomicCity variable, 193–196
Attendee class, 308–310
autoboxing, 321–323
AWT (Abstract Window Toolkit), 428

B

backslash (\), 245
Backus, John, 329
balance variable, 158–159
baseball players
  creating package for, 282–284
  finding team batting average, 301–307
  formatting, 289–290
  instances in, 288–289
  running code, 287–288
  simple source file for, 285–286
  using package for, 284–287
BigDecimal class, 178–179
birth method, 198–200
bits, 63
blocks, 83, 103, 151–152
books about Java, 450–451
boolean type, 78–79
Boyarsky, Jeanne, 450–451

break statements, 130, 152–153
Bright, Herbert, 377–378
bugs in programs
  ArithmeticException, 392, 393
  catching exceptions, 380, 382
  catching two or more exceptions at once, 394
  checked exceptions, 396
  custom-made exceptions, 385–387
  errors versus exceptions, 394
  FileNotFoundException, 244–245, 395–399, 400–401
  general discussion, 378–381
  InputMismatchException, 241, 288
  matching catch clause, 387–393
  NumberFormatException, 379–381, 384
  NumberTooLargeException, 389–394
  OutOfRangeException, 386–387, 388, 389–394
  parameters in catch clauses, 382–385
  runtime exceptions, 386–387, 400
  throwing exceptions, 378–381
  throws clauses, 399, 401
  try–catch statements, 396–398, 401
  try-with-resources feature, 402–405
  unchecked exceptions, 396
byte type, 69, 75
bytecode, 25, 26–28, 30

C

C programming language, 11
C++ programming language, 11
calculating interest
  experiments with, 177
  general discussion, 171–173
  passing values to methods, 174–175
  returning values from, 175–177
calling methods, 94–95
case sensitivity, 40, 49
Cat class, 421
catch clause
  catching two or more exceptions at once, 394
  matching, 387–393
  overview, 380
  parameters in, 382–385
  try–catch statements, 382, 384–385, 396–398, 401
  try-with-resources feature, 404–405
char type, 75–77, 148–149
charAt method, 149
ChatGPT, 9
checked exceptions, 396
City class, 201
classes
  abstract
    abstract methods in, 424–425
    creating art, 423–424
    extending, 420–422
    general discussion, 417–419
    testing, 422–423
  access for members in different, 295–299
  access for members in same, 295
  access modifiers for, 293–294
  Account
    accessor methods, 180–186
    calculating interest, 171–173
    declaring variables and creating objects, 160–161
    display method, 164–171, 186
    displaying accounts, 164–171
    fields in, 158
    getInterest method, 173–177, 186
    passing values to methods, 174–175
    returning values from, 175–177
    using object fields, 161–162
  AirplaneFlight, 219–220, 267
  Animal, 418
  Apartment, 261, 266
  ArrayList
    add methods, 317
    arrays versus, 356–357
    autoboxing, 321–323
    enhanced for loop, 318–319, 324
    experiments with, 329
    general discussion, 314–317
    getting streams from, 336
    hasNext method, 323–324
    importing module, 319–320
    iterators, 323–324
    next method, 324
    overview, 328
    using generics, 320–321
  Arrays, 364

classes *(continued)*
Attendee, 308–310
BigDecimal, 178–179
calling static methods, 307
Cat, 421
City, 201
collection
experiments with, 329
general discussion, 328
HashMap class, 326–328
overview, 313
sorting and searching, 324–326
ColumnOfNumbers, 411–412
creating objects, 160–161
declaring variables, 160–161
defining methods in, 164–171
Dog, 421
Double, 322
Employee, 15–16, 230–235
EnglishSpeakingWorld, 190–193
EnglishSpeakingWorld2, 193–196
experiments with, 164
File, 149–151, 243–244
FoodProduct, 164, 171, 177, 186
Fork, 275–276
FullTimeEmployee
creating, 253–257
using, 258–261
general discussion, 158–160
HashSet, 328
Hotel.class, 27
House, 261, 266
HousePet, 418–424
implicit, 185–190
inner, 294
Integer, 186, 321–323
java.io.PrintStream, 359
javax.swing.JOptionPane, 118–119
JFrame, 37
JOptionPane, 114, 118–119
LinkedList, 328, 362
LocalTime, 219
Math, 223–224
members of, 167
NonProfitOrganization, 261
NumberFormatException, 379–381, 384
Numbers, 417
in object-oriented programming, 15–16
Organization, 164, 171, 177, 186
Organization2, 261
OutOfRangeException, 386–387, 388, 389–394
Painting, 299
PartTimeEmployee
creating, 253, 257–258
using, 258–261
PlaceToLive, 237, 247, 261
Player
creating ArrayList of objects, 329
creating package, 282–284
finding team batting average, 302–307
formatting, 289–290
instances in, 288–289
running code, 287–288
simple source file for, 285–286
using package, 284–287
PlayingCard, 291, 311
PriorityQueue, 328
ProfitMakingOrganization, 261
public, 189, 293–294
Queue, 328
Random
calling methods with, 94–95
creating and referring to instances with, 93
experiments with, 95–96
if statements and, 99
overview, 91
reserving variable names with, 91
record
equals method, 226–228
experiments with, 228
general discussion, 220–223
overview, 303
switch structures with, 223–225
Room, 367–370, 371
RuntimeException, 400
Scanner, 234–237, 244, 323, 351–352, 395–396
sealed, 294
Shape, 423–424
ShowPlayers, 286, 308
Song, 299

Square, 423–424
Stack, 328
String
calling methods with, 94–95
charAt method, 149
concatenating values, 80
creating and referring to instances with, 93
equals method, 108–110
experiments with, 95–96
format method, 177–180, 186
overview, 77, 91
reserving variable names with, 91
Student, 219, 371
subclasses
annotations, 263–264
constructors for, 272–273, 274–278
creating, 253–258
experiments with, 266–267
matching types, 259–260
overriding methods, 262–266
overview, 250–253
switch's pattern-matching feature, 268–270
for Temperature class, 270–272, 273–274
using, 258–261
superclass, 254
Table, 417
Temperature
accessor methods, 208–211
creating colder temperatures, 223–225
default constructor, 217–218
record classes, 220–223
subclasses for, 270–272, 273–274
use of constructors, 215–217
TemperatureCelsius, 277–278
TemperatureForPercy, 270–274
Triangle, 423–424
UseAccount, 187–189
using object fields, 161–162
using several in programs, 162
var declarations and, 163
Work, 299
wrapper, 321–323
close method, 236
COBOL programming language, 10
code
benefits of writing, 9–10
comments
commenting out code, 54–55
end-of-line, 52
javadoc, 52
knowing when to add, 53–54
overview, 50
traditional, 51–52
types of, 51–53
compiling, 32–33
copying and pasting, 242–243
Java source code, 26
modifying, 32–33, 340–341
overview, 9
pseudocode, 14
reusing
annotations, 263–264
constructors for, 272–273, 274–278
creating subclasses, 253–258
experiments with, 266–267
matching types, 259–260
overriding methods, 262–266
overview, 249–250
switch's pattern-matching feature, 268–270
for Temperature class, 270–272, 273–274
using subclasses, 258–261
running, 32–33
safe, 181
simple, 181
VS Code, 22
Coderanch, 447
collection classes
ArrayList
add methods, 317
autoboxing, 321–323
enhanced for loop, 318–319
general discussion, 314–317
hasNext method, 323–324
importing module, 319–320
iterators, 323–324
next method, 324
using generics, 320–321
experiments with, 329
general discussion, 328
HashMap class, 326–328
overview, 313
sorting and searching, 324–326

ColumnOfNumbers class, 411–412
combining comparisons in parentheses, 116–117
com.example.baseball package, 282–283
comments
  commenting out code, 54–55
  end-of-line, 52
  javadoc, 52
  knowing when to add, 53–54
  overview, 50
  traditional, 51–52
  types of, 51–53
compact constructors, 303–304
compareToIgnoreCase method, 329
comparison operators, 107–108
compiler, 25–28
compiling code, 32–33
components of arrays, 350
computer programming languages
  C, 11
  C++, 11
  COBOL, 10
  FORTRAN, 10, 329, 377–378
  history of, 10–12
  Java development, 11–12
  LISP, 10, 330
  overview, 9
  Simula 67, 11
  which to learn, 451
computers, benefits of using, 7–8
concatenating String values, 80
conditional operator, 370–371
conditions
  comparison operators, 107–108
  equals method, 108–110
  if statements
    accessor methods and, 185
    controlling keystrokes from keyboard, 99–100
    curly braces in, 103
    double equal signs, 102–103
    double values, 100–101
    experiments with, 105–107
    general discussion, 98–99, 101–102
    indenting, 103–104
    nested, 122–123
    without else part, 103–104
  import declarations, 117–120
  logical operators, 112–114
  null in, 114–116
  in parentheses, 116–117
  test blocks, 111–112
constructors
  calling, 150, 161, 211–214
  compact, 303–304
  declaring, 208–211
  default, 217–219
  enum type, 207–208
  equals method and, 226–228
  experiments with, 219–220, 228
  inherited toString method, 214
  overview, 206
  parameter lists, 215–217
  record classes, 220–223
  for subclasses, 272–273, 274–278
  switch structures with, 223–225
continue statements, 153
cookedBooks.txt file, 148
copying and pasting code, 242–243
counting boxes program, 378–384
counting loops, 139–141
country standards for number formats, 101, 106, 239–240, 288
Cross Reference icon, 3
curly braces ({}), 46–47, 103, 104, 151
currency values, 177–180, 370
custom-made exceptions, 385–387
cutCheck method, 232, 397–400, 404–405

D

data sharing, 341–343
decimal numbers, 105
decision-making statements
  combining comparisons in parentheses, 116–117
  comparison operators, 107–108
  equals method, 108–110
  if statements
    controlling keystrokes from keyboard, 99–100
    curly braces in, 103
    double equal signs, 102–103
    double values, 100–101
    experiments with, 105–107

general discussion, 98–99, 101–102
indenting, 103–104
without `else` part, 103–104
import declarations, 117–120
logical operators, 112–114
nested `if` statements, 122–123
`null` in, 114–116
`switch` statements
experiments with, 132–133
with fall-through, 130–132
general discussion, 123–127
old-style, 129–130
`switch` expressions, 127–129
test blocks, 111–112
declarations
import, 117–120, 168–170, 286–287, 306–307
import module, 315–317
`import static`, 306–307
method, 42–43
package, 283
`var`, 70, 149–150, 163, 350
variable, 64, 69
declarative programming, 435
declaring variables, 83
decrement operators, 83–88
default access, 185–186, 298
default constructors, 217–219
`default` keyword, 415–416
default methods, 415–416
`Deletable` interface, 416–417
`delete` method, 151
deleting files, 146–148
developing programs, 32–33
`diskScanner.close()` call, 247
`display` method, 164–171, 186, 410, 412
`Displayable` interface, 409–410, 412–413
`displayMe` method, 414–415
division operator, 81
`do` loop, 145–147, 152
`doBirth` method, 202–203
`Dog` class, 421
dot notation, 167
double backslash (\\), 245
`Double` class, 322
double equal signs (==), 102–103, 107, 108
double plus signs (++), 83–88
double quotes (""""), 111
`double` type, 64–66, 75, 100–101, 322
`duration` method, 267

E

Eclipse, 22
elements of arrays, 350
`else` statements, 103–104
emojis, 105
`Employee` class, 15–16, 230–235
`EmployeeInfo.txt` file, 239, 241–242, 244–245
`EmployeePayment.txt` file, 395–400
employees
about, 230–232
changing payments for some, 262–266
creating subclasses for, 253–258
`Employee` class, 232–235
exception handling for payroll checks, 395–400
full-time versus part-time, 250–253
`switch`'s pattern matching, 268–270
using subclasses for, 258–261
writing payroll checks, 237–242
end-of-line comments, 52
enforcing rules, with accessor methods, 185–186
English language, limitations in computer programming, 8–9
`EnglishSpeakingWorld` class, 190–193
`EnglishSpeakingWorld2` class, 193–196
enhanced `for` loop, 152, 318–319, 324, 353–355
`enum` type, 207–208, 224–225
`equals` method, 108–110, 226–228
`equalsIgnoreCase` method, 110
errors, 394
escape sequences, 352–353
`evidence` variable, 151
exception handling
`ArithmeticException`, 392, 393
catching exceptions, 380, 382
catching two or more exceptions at once, 394
checked exceptions, 396
custom-made exceptions, 385–387
errors versus exceptions, 394
`FileNotFoundException`, 244–245, 395–399, 400–401
general discussion, 378–381

exception handling *(continued)*
InputMismatchException, 241, 288
matching catch clause, 387–393
NumberFormatException, 379–381, 384
NumberTooLargeException, 389–394
OutOfRangeException, 386–387, 388, 389–394
parameters in catch clauses, 382–385
runtime exceptions, 386–387, 400
throwing exceptions, 378–381
throws clauses, 399, 401
try–catch statements, 396–398, 401
try-with-resources feature, 402–405
unchecked exceptions, 396
exclamation point (!), 144
exists method, 151
explicit import declaration, 120
expressions, 78–79, 141–142
eXtensible Markup Language (XML) tags, 435–436

F

factorials, 144
fall-through, 130–132
fields
in Account class, 158–159
defined, 159
initializing in class declarations, 163
versus method-local variables, 191–193
of objects, using, 161–162
private, 183–185, 210–211, 257
static
collection classes and, 326
compact constructors, 303–304
experiments with, 310–312
general discussion, 301–303
import static declaration, 306–307
using with other methods, 308–310
File class, 149–151, 243–244
file-handling statements, 150–151
FileNotFoundException, 244–245, 395–399, 400–401
files
AllWords.txt, 315–317
for classes, 162
cleaning up, 247
connecting arrays to, 360–361
cookedBooks.txt, 148
copying and pasting code, 242–243
deciding where to put, 245
deleting, 146–148
disk input with Scanner class, 234–237
EmployeeInfo.txt, 239, 241–242, 244–245
EmployeePayment.txt, 395–400
File class, 149–151, 243–244
file-handling statements, 150–151
finding, 244–245
GuestList.txt, 351–352, 360–361
Hankies.txt, 287–288, 290
HelloApplication.java, 434
HelloController.java, 434, 440–441
hello–view.fxml, 434, 435–436, 437, 440
Hotel.class, 27
Main.java, 22–23
module–info.java, 292–293
NotFoundException, 244–245, 395–399, 400–401
reading by line, 245–247
reading characters from, 243–244
reading keystrokes from keyboard, 234–237
RoomList.txt, 367
simple source, 170, 187–189, 285–286
WordFrequencies.txt, 326–327
writing arrays to, 359–360
filter method, 18, 336–337
final keyword, 381
findPaymentAmount method, 263–264
findTeamAverage method, 303
float type, 64–66, 75
folder, root, 148
FoodProduct class, 164, 171, 177, 186
for statements/loops
"Al's All Wet" song, 142–143
anatomy of, 141–142
enhanced for loop, 152, 318–319, 324, 353–355
experiments with, 143–145
general discussion, 139–141
forEach method, 17, 345
Fork class, 275–276
format method, 177–180, 186
formatting
country standards, 101, 106, 239–240, 288
numbers, 177–180
in packages, 289–291
FORTRAN programming language, 10, 329, 377–378

FullTimeEmployee class
  creating, 253–257
  using, 258–261
fully parameterized constructor, 218
functional programming
  data sharing, 341–343
  experiments with, 346–347
  general discussion, 17–19, 329–332
  lambda expressions, 332–334
  LISP, 10
  method references, 343–344
  modifying code, 340–341
  side effects in, 344–345
  streams, 334–340

G

Garden State Java User Group, 446
generating web pages from javadoc comments, 52–53
generics, 320–321
get method, 327
getInterest method, 173–177, 186
getMessage method, 383
getter methods, 184
goToTheSupermarketAndBuySome method, 171–177
GPT tools, 9
Grazi, Victor, 450–451
greater than operator (>), 107
greater than or equal to operator (>=), 107
grouping separators, 234, 239–241
Guessing Game example, 98–99, 190
GuestList.txt file, 351–352, 360–361
GUI (graphical user interface)
  classes for creating, 428
  installing JavaFX plugin, 429
  installing Scene Builder, 430–431
  JavaFX project
    adding visual elements, 436–442
    creating, 431–433
    running, 434–435
  setting up computer, 428–429

H

Hankies.txt file, 287–288, 290
Harvard Mark II computer, 377
HashMap class, 326–328
HashSet class, 328
hasNext method, 323–324
Hello world! program, 28–30
HelloApplication.java file, 434
HelloController.java file, 434, 440–441
hello-view.fxml file, 434, 435–436, 437, 440
history of programming languages, 10–12
Hopper, Grace, 10
Hotel.class file, 27
House class, 261, 266
HousePet class, 418–424
howToCareFor method, 419, 420–423

I

icons, used in book, 3
IDE (integrated development environment)
  commenting out code, 55
  creating packages, 284
  installing, 22
  program arguments, 372–374
identifiers, 37–38
if statements
  accessor methods and, 185
  controlling keystrokes from keyboard, 99–100
  curly braces in, 103
  double equal signs, 102–103
  double values, 100–101
  experiments with, 105–107
  general discussion, 98–99, 101–102
  indenting, 103–104
  nested, 122–123
  without else part, 103–104
imperative programming, 13–14, 329–330, 341–342, 435
implements keyword, 412
implicit classes, 185–190
implicit import declaration, 120, 169–170
import declarations, 117–120, 168–170, 286–287, 306–307
import module feature, 315–317, 319–320
import static declaration, 306–307
inaccessible fields, creating, 183–185
increment operator, 83–88
indenting
  if statements, 103–104
  lines of code, 46, 47–48, 49

InfoQ website, 447
inherited `toString` method, 214
initialization, `for` statements, 141–142
initializing variables, 69–71, 83
inner classes, 294
`InputMismatchException`, 241, 288
inserting values into arrays, 361–362
installing
  IDE, 22
  Java Development Kit, 21
  JavaFX plugin, 429
  Scene Builder, 430–431
instance variable, 191
instances
  in object-oriented programming, 17, 158
  in packages, 288–289
`int` type, 69, 75, 81, 99–100, 321–323
`Integer` class, 186, 321–323
`Integer.parseInt` method, 100, 109
integrated development environment (IDE)
  commenting out code, 55
  creating packages, 284
  installing, 22
  program arguments, 372–374
IntelliJ IDEA
  adding visual elements to JavaFX app, 436–442
  creating JavaFX app, 431–433
  finding Scene Builder, 436–437
  installing JavaFX plugin, 429
  overview, 22, 23–24, 428–429
  running JavaFX app, 434–435
interest, calculating
  experiments with, 177
  general discussion, 171–173
  passing values to methods, 174–175
  returning values from, 175–177
interfaces
  abstract methods in, 424–425
  declaring two, 409–411
  default methods, 415–416
  general discussion, 408–409
  implementing, 411–413
  power of, 413–415
intermediate operation, 340
Internet resources
  for book, 445
  Coderanch, 447
  Garden State Java User Group, 446
  InfoQ, 447
  Java, 446
  Java API documentation, 446
  Java Community Process, 447
  Java language specification, 446
  New York JavaSIG, 446
  news about Java, 446
  Oracle, 446
  Stack Overflow, 447
  for technical questions, 447
  TheServerSide, 447
  TIOBE Programming Community Index, 451
  user groups, 446
inventory program
  catching exceptions, 382–384
  catching two or more exceptions at once, 394
  custom-made exceptions, 385–387
  matching `catch` clause, 387–393
  throwing exceptions, 378–381
iterations, 138
iterators, 323–324

J

Java
  case sensitivity in, 40, 49
  general discussion, 8–10
  history of, 10–12
  learning, 450–451
  user groups, 446, 452–453
  website, 446
Java API documentation, 36–37, 119, 381, 446
Java bytecode, 25, 26–28, 30
Java Bytecode Editor, 27
Java Community Process (JCP), 37, 447
Java Development Kit (JDK)
  compiler, 25–28
  downloading, 446
  installing newest release of, 21–22
  Java virtual machine, 28–32
  overview, 24

Java interfaces
abstract methods in, 424–425
declaring two, 409–411
default methods, 415–416
general discussion, 408–409
implementing, 411–413
power of, 413–415
Java Language Specification, 36, 446
Java Motel
booking hotel guests, 351–352
checking for vacancies, 357–359
creating array, 349–351
`Room` class, 367–370
Java resources
for book, 445
books, 450–451
Coderanch, 447
Garden State Java User Group, 446
InfoQ, 447
Java, 446
Java API documentation, 446
Java Community Process, 447
Java language specification, 446
New York JavaSIG, 446
news about Java, 446
Oracle, 446
Stack Overflow, 447
for technical questions, 447
TheServerSide, 447
user groups, 446
websites, 445–447
Java source code, 26
Java Specification Requests (JSRs), 447
Java Swing, 428
Java virtual machine (JVM), 25, 28–32
`java.base`, implicit importing for, 170
javadoc comments, 52–53
JavaFX
adding visual elements to app, 436–442
creating app, 431–433
installing plugin, 429
installing Scene Builder, 430–431
overview, 114, 428
running app, 434–435
setting up computer, 428–429
`java.io` package, 170
`java.io.PrintStream` class, 359
`java.util.Arrays` package, 364
`javax.swing.JOptionPane` class, 118–119
JCP (Java Community Process), 37, 447
JDK (Java Development Kit)
compiler, 25–28
downloading, 446
installing newest release of, 21–22
Java virtual machine, 28–32
overview, 24
JetBrains, 429
`JFrame` class, 37
JJava Jupyter kernel, 74
`JOptionPane` class, 114, 118–119
JShell, 71–74
JSRs (Java Specification Requests), 447
JUnit, 177
JVM (Java virtual machine), 25, 28–32

K

keyboard
controlling keystrokes from, 99–100
reading keystrokes from, 234–237
`keyboard.close()` statement, 236
keywords
`abstract`, 419
`catch`
matching, 387–393
overview, 380
parameters in, 382–385
`try–catch` statements, 382, 384–385, 396–398, 401
try-with-resources feature, 404–405
`default`, 415–416
exception handling, 380
`final`, 381
general discussion, 37–39
`implements`, 412
`private`, 183–184
`super`, 273, 274, 419
`this`, 167, 193, 216–217
`throw`, 380
`throws`, 380, 396, 399, 401
`try`, 380, 382, 387–393, 396

L

lambda expressions
  general discussion, 332–334
  overview, 343
  side effects, 344–345
languages, programming
  C, 11
  C++, 11
  COBOL, 10
  FORTRAN, 10, 329, 377–378
  history of, 10–12
  Java development, 11–12
  LISP, 10, 330
  overview, 9
  Simula 67, 11
  which to learn, 451
`largestSoFar` variable, 355–356
less than operator (<), 107
less than or equal to operator (<=), 107
line of output, leaving cursor at end of, 66
`LinkedList` class, 328, 362
Linux operating system, program for, 28–29
Lisp programming language, 10, 330
literals, 61
`LocalTime` class, 219
logical operators
  general discussion, 112–114
  `null` and, 114–116
  parentheses and, 116–117
`long` type, 69, 75
long-term support (LTS) releases, 12
loops
  blocks, 151–152
  counting, 139–141
  `do` loop, 145–147
  enhanced `for`, 152, 318–319, 324, 353–355
  experiments with, 152–153
  file-handling statements, 150–151
  iterations, 138
  overview, 135
  reading single characters, 148–149
  root folder, 148
  `for` statements, 139–145
  `var` declarations, 149–150
  `while` statements, 136–139
LTS (long-term support) releases, 12

M

`main` method
  creating `Employee` instances, 232–233
  creating objects, 160
  general discussion, 43–44, 45
  implicitly declared classes, 187–188
  initializing variables in, 70
  simplified code for, 233–234
  `static` keyword and, 308–310
  traditional versus easy-to-read, 212–213
`Main.java` file, 22–23
`map` method, 337
Mariner I, 378
`mars` variable, 191–193, 194–196
matching `catch` clause, 387–393
matching types, 259–260
`Math` class, 223–224
`max` method, 223–224
McCarthy, John, 10, 330
members of class, 167
method body, 43
method call, 43
method declarations, 42–43
method headers, 43
method-local variables, 191–193, 294–295
methods
  abstract, 411, 415, 419, 424–425
  accessor
    enforcing rules with, 185–186
    general discussion, 180–183
    making fields inaccessible, 183–185
    `Temperature` class, 208–211
  `add`, 317
  `Arrays.toString`, 362
  `birth`, 198–200
  calling, 94–95
  calling with reference types, 94–95
  `charAt`, 149
  `close`, 236

compareToIgnoreCase, 329
cutCheck, 232, 397–400, 404–405
default, 415–416
defining in classes, 164–171
delete, 151
display, 164–171, 186, 410, 412
displayMe, 414–415
doBirth, 202–203
duration, 267
equals, 108–110, 226–228
equalsIgnoreCase, 110
exists, 151
experiments with, 177
filter, 336–337
findPaymentAmount, 263–264
findTeamAverage, 303
forEach, 17, 345
format, 177–180, 186
general discussion, 40–43, 171–173
get, 327
getInterest, 173–177, 186
getMessage, 383
getter, 184
goToTheSupermarketAndBuySome, 171–177
hasNext, 323–324
howToCareFor, 419, 420–423
inherited toString, 214
Integer.parseInt, 100, 109
main
  creating Employee instances, 232–233
  creating objects, 160
  general discussion, 43–44, 45
  implicitly declared classes, 187–188
  initializing variables in, 70
  simplified code for, 233–234
  static keyword and, 308–310
  traditional versus easy-to-read, 212–213
map, 337
max, 223–224
next, 235, 324
nextDouble, 235, 246, 247
nextInt, 94–95, 235
nextLine, 236, 245–247
overriding with subclasses, 262–266
parseInt, 186
passing values to methods, 174–175
peek, 346
println, 45, 46, 79, 103, 127–128
printStackTrace, 383–384
put, 327–328
readInt, 100
readln, 148–149, 236
reduce, 18, 337–340, 341
references in functional programming, 343–344
replace, 94
returning objects from, 201–203
returning values from, 175–177
setName, 184–185
setter, 184
size, 329
sort, 363–364
start, 434
static
  collection classes and, 326
  compact constructors, 303–304
  experiments with, 310–312
  general discussion, 301–303
  import static declaration, 306–307
  using with other methods, 308–310
String.format, 289–290, 307
sum, 343
summarize, 410–411, 412
summarizeMe, 415
System.out.println, 143–144, 189
toString, 222–223
visitIdaho, 194–196
visitNewJersey, 195–196
visitPennsylvania, 191–193
minus sign, 80
modifiers, access
  for classes, 293–294
  for members in classes, 295–299
  overview, 291, 292
modifying code, 32–33, 340–341
module-info.java file, 292–293
modules, 119, 292–293, 319–320
multicore processors, modifying code for, 340–341
multiplication operator, 81
myAccount variable, 158–159, 161, 162, 167

N

name variable, 158–159
naming packages, 282–284
nested if statements, 122–123
NetBeans, 22
networking, 453
new Account(), 161, 162
New Project dialog box, IntelliJ IDEA, 431–432
New York JavaSIG website, 446
next method, 235, 324
nextDouble method, 235, 246, 247
nextInt method, 94–95, 235
nextLine method, 236, 245–247
NonProfitOrganization class, 261
not equal to operator (!=), 107, 108
not operator (!), 112
null, 114–116
number field, 216–217
NumberFormatException, 379–381, 384
numbers
  country formats, 101, 106, 239–240, 288
  format experiments, 66–67
  NumberFormatException, 379–381, 384
  NumberTooLargeException, 389–394
  storing whole, 69
  types of values in variables, 63–66
  without decimal points, 67–69
Numbers class, 417
NumberTooLargeException, 389–394

O

object fields, 161–162
object-oriented programming (OOP)
  accessor methods, 180–186
  classes, 158–160, 162
  creating objects, 160–161
  declaring variables, 160–161
  defining methods in classes, 164–171
  fields versus method-local variables, 191–193
  format strings, 177–180
  general discussion, 14–17
  implicit classes, 185–190
  modifying code, 250
  object fields, 161–162
  overview, 157
  parameter lists, 197–203
  passing values to methods, 174–175
  returning values from methods, 175–177
  Simula 67, 11
  String and Random classes, 92–96
  telling variables where to go, 193–196
  var declarations, 163
objects
  in Account class, 158
  arrays of, 365–371
  in object-oriented programming, 15–17
  passing to methods, 199–200
  returning from methods, 201–203
Online icon, 3
operators
  assignment, 88–90
  asterisk, 81
  comparison, 107–108
  conditional, 370–371
  declaring variables, 83
  increment and decrement, 83–88
  logical, 112–114
  minus sign, 80
  plus sign, 79–80
  remainder operator, 81–82
  slash, 81
or operator (||), 112, 116–117
Oracle Corporation, 11–12
Oracle website, 446
Organization class, 164, 171, 177, 186
Organization2 class, 261
outline forms, 46–50
OutOfRangeException, 386–387, 388, 389–394
@Override annotation, 263–264, 413
overriding methods, 262–266

P

package declaration, 283
packages
  access in, 298
  formatting in, 289–291
  instances, 288–289
  modules, 292–293
  naming, 282–284

overview, 119, 282
subpackages, 287, 298
unnamed, 283
using Player class, 284–287
package-wide access, 298
Painting class, 299
paradigm, 13
parallel stream, 341
parameters/parameter lists
in catch clauses, 382–385
constructor calls, 213–214, 215–217
lambda expressions and, 333
naming parameters, 217
overview, 46
passing by reference, 199–201
passing by value, 197–198
returning objects from methods, 201–203
returning results, 198–199
sending values to and from methods, 171
type matching, 259–260
parentheses, conditions in, 116–117
parseInt method, 186
PartTimeEmployee class
creating, 253, 257–258
using, 258–261
PartTimeWithOver subclass, 262–263
passing
parameter lists by reference, 199–201
parameter lists by value, 197–198
values to methods, 174–175
passwords, 109–110, 112–114, 121
pattern-matching feature, of switch structure, 268–270
payment for employees, changing, 262–266
payroll checks, writing, 237–242
peek method, 346
Pentium processor, program for, 28–29
PlaceToLive class, 237, 247, 261
Player class
creating ArrayList of objects, 329
creating package, 282–284
finding team batting average, 302–307
formatting, 289–290
instances in, 288–289
running code, 287–288
simple source file for, 285–286
using package, 284–287
playerCount field, 303
PlayingCard class, 291, 311
plus sign, 79–80
Pope, Alexander, 145
postdecrement operator, 85–86
postincrement operator, 84–85, 86–88
PowerPC processor, program for, 29–30
predecrement operator, 85–86
preincrement operator, 83–84, 86–88
preview features, 22–23, 40
primitive types
ArrayList class and, 321–323
arrays and, 357
boolean type, 78–79
char type, 75–77
overview, 74–75, 407
perimeter passing, 198
wrapper classes, 322
print calls, 66–67, 103, 289–290
println method, 45, 46, 79, 103, 127–128
printStackTrace method, 383–384
PrintStream, 359–361
PriorityQueue class, 328
private access, 295, 296
private fields, 183–184, 186, 210–211, 257
private keyword, 183–184
ProfitMakingOrganization class, 261
program arguments, 371–375
program bugs
ArithmeticException, 392, 393
catching exceptions, 380, 382
catching two or more exceptions at once, 394
checked exceptions, 396
custom-made exceptions, 385–387
errors versus exceptions, 394
FileNotFoundException, 244–245, 395–399, 400–401
general discussion, 378–381
InputMismatchException, 241, 288
matching catch clause, 387–393
NumberFormatException, 379–381, 384
NumberTooLargeException, 389–394
OutOfRangeException, 386–387, 388, 389–394
parameters in catch clauses, 382–385
runtime exceptions, 386–387, 400
throwing exceptions, 378–381

program bugs *(continued)*
throws clauses, 399, 401
try–catch statements, 396–398, 401
try-with-resources feature, 402–405
unchecked exceptions, 396
programming
declarative, 435
functional
data sharing, 341–343
experiments with, 346–347
general discussion, 17–19, 329–332
lambda expressions, 332–334
LISP, 10
method references, 343–344
modifying code, 340–341
side effects in, 344–345
streams, 334–340
imperative, 13–14, 329–330, 341–342, 435
OOP
accessor methods, 180–186
classes, 158–160, 162
creating objects, 160–161
declaring variables, 160–161
defining methods in classes, 164–171
fields versus method-local variables, 191–193
format strings, 177–180
general discussion, 14–17
implicit classes, 185–190
modifying code, 250
object fields, 161–162
overview, 157
parameter lists, 197–203
passing values to methods, 174–175
returning values from methods, 175–177
Simula 67, 11
String and Random classes, 92–96
telling variables where to go, 193–196
var declarations, 163
programming languages
C, 11
C++, 11
COBOL, 10
FORTRAN, 10, 329, 377–378
history of, 10–12
Java development, 11–12
LISP, 10, 330
overview, 9
Simula 67, 11
which to learn, 451
programs
compiler, 25–28
developing, 32–33
Java virtual machine, 28–32
overview, 9
Project tool window, IntelliJ IDEA, 437
protected access, 298–299
pseudocode, 14
public access for members of classes, 297–298
public classes, 189–190, 293–294
public fields, 184, 298
put method, 327–328

Q

Queue class, 328
quotation marks, 49

R

Random class
calling methods with, 94–95
creating and referring to instances with, 93
experiments with, 95–96
if statements and, 99
overview, 91
reserving variable names with, 91
Read Evaluate Print Loop (REPL), 71
reading
characters from files, 243–244
keystrokes from keyboard, 234–237
one line at a time, 245–247
single characters, 148–149
readInt method, 100
readln method, 148–149, 236
Real World Java (Boyarsky & Grazi), 450–451
record classes
equals method, 226–228
experiments with, 228
general discussion, 220–223
Player class, 303

  switch structures with, 223–225
reduce method, 18, 337–340, 341
reference, passing parameter lists by, 199–201
reference types. *See also* arrays; classes
  abstract classes, 417–424
  abstract methods, 424–425
  ArrayList class and, 321
  calling methods with, 94–95
  creating and referring to instances with, 93
  experiments with, 95–96
  general discussion, 407–408
  interfaces
    declaring two, 409–411
    general discussion, 408–409
    implementing, 411–413
    power of, 413–415
  overview, 90–91
  Random class, 91
  reserving variable names with, 91
  String class, 91
remainder operator, 81–82
Remember icon, 3
repeating guessing game, 136–139
REPL (Read Evaluate Print Loop), 71
replace method, 94
reply variable, 151–152
resources
  for book, 445
  books on Java, 450–451
  Coderanch, 447
  Garden State Java User Group, 446
  InfoQ, 447
  Java API documentation, 446
  Java Community Process, 447
  Java language specification, 446
  Java website, 446
  New York JavaSIG, 446
  news about Java, 446
  Oracle, 446
  Stack Overflow, 447
  for technical questions, 447
  TheServerSide, 447
  TIOBE Programming Community Index, 451
  user groups, 446
return statement, 175–176, 199
return type, 171
returning
  objects from methods, 201–203
  parameter lists results, 198–199
  values from methods, 175–177
reusing code
  annotations, 263–264
  constructors for, 272–273, 274–278
  creating subclasses, 253–258
  experiments with, 266–267
  matching types, 259–260
  overriding methods, 262–266
  overview, 249–250
  switch's pattern-matching feature, 268–270
  for Temperature class, 270–272, 273–274
  using subclasses, 258–261
Ritchie, Dennis, 11
Room class, 367–370, 371
RoomList.txt file, 367
root folder, 148
run configuration, 372
running code, 32–33
runningTotal variable, 356
runtime exception, 386–387, 400

S

safe code, 181
salami scam, 177
Scanner class, 234–237, 244, 323, 351–352, 395–396
Scene Builder
  building scenes, 437–439
  finding on computer, 436–437
  installing, 430–431
sealed classes, 294
searching, in collection classes, 324–326
seeing stars, 144–145
semicolons, 46, 49, 73
serial stream, 341
setName method, 184–185
setter methods, 184
Shape class, 423–424
short type, 69, 75
showMessageDialog call, 114, 116
ShowPlayers class, 286, 308

side effects, 344–345
simple code, 181
simple source files, 170, 187–189, 285–286
Simula 67 programming language, 11
single characters, reading, 148–149
`size` method, 329
slash (/) operator, 81
`smackoverAR` variable, 200–202
`smackoverARpop` variable, 198–199
software
  compiler, 25–28
  developing, 32–33
  installing required, 21–23
  Java virtual machine, 28–32
  overview, 9
software developers, advice for, 449–454
software engineering, 451
`Song` class, 299
`sort` method, 363–364
sorting, in collection classes, 324–326
`Square` class, 423–424
`Stack` class, 328
Stack Overflow, 447
`start` method, 434
statements
  assignment, 62
  `break`, 130, 152–153
  combining comparisons in parentheses, 116–117
  comparison operators, 107–108
  `continue`, 153
  `else`, 103–104
  `equals` method, 108–110
  file-handling, 150–151
  `for`
    "Al's All Wet" song, 142–143
    anatomy of, 141–142
    enhanced `for` loop, 152
    experiments with, 143–145
    general discussion, 139–141
  general discussion, 44–46
  `if`
    accessor methods and, 185
    controlling keystrokes from keyboard, 99–100
    curly braces in, 103
    double equal signs, 102–103
    `double` values, 100–101
    experiments with, 105–107
    general discussion, 98–99, 101–102
    indenting, 103–104
    nested, 122–123
    without `else` part, 103–104
  import declarations, 117–120
  `keyboard.close()`, 236
  logical operators, 112–114
  nested `if`, 122–123
  `null` in, 114–116
  `return`, 175–176, 199
  `switch`
    `enum` types and, 224–225
    experiments with, 132–133
    with fall-through, 130–132
    general discussion, 123–127
    `for` loops and, 143
    old-style, 129–130
    pattern-matching feature, 268–270
    `switch` expressions, 127–129
  test blocks, 111–112
  `try-catch`, 382, 384–385, 396–398, 401
  try-with-resources, 402–405
  `while`
    `break` statements and, 152–153
    `continue` statements and, 153
    general discussion, 136–139
    overview, 145–146
`static` fields and methods
  collection classes and, 326
  compact constructors, 303–304
  experiments with, 310–312
  general discussion, 301–303
  `import static` declaration, 306–307
  using with other methods, 308–310
streams
  filtering items in, 336–337
  general discussion, 334–335
  getting from collection, 336
  getting values from items in, 337
  summarizing items in, 337–340

String class
  calling methods with, 94–95
  charAt method, 149
  concatenating values, 80
  creating and referring to instances with, 93
  equals method, 108–110
  experiments with, 95–96
  format method, 177–180, 186, 289–290, 307
  overview, 77, 91
  reserving variable names with, 91
Stroustrup, Bjarne, 11
Student class, 219, 371
subclasses
  annotations, 263–264
  constructors for, 272–273, 274–278
  creating, 253–258
  experiments with, 266–267
  matching types, 259–260
  overriding methods, 262–266
  overview, 250–253
  switch's pattern-matching feature, 268–270
  for Temperature class, 270–272, 273–274
  using, 258–261
subpackages, 287, 298
sum method, 343
Summarizable interface, 410–411, 412–413
summarize method, 410–411, 412
summarizeMe method, 415
Sun Microsystems, 11
super keyword, 273, 274, 419
superclass, 254
surrogates, 105
Swing, 428
switch expressions, 127–129, 223–225
switch statements
  enum types and, 224–225
  experiments with, 132–133
  with fall-through, 130–132
  general discussion, 123–127
  for loops and, 143
  old-style, 129–130
  pattern-matching feature, 268–270
  switch expressions, 127–129
System.in, 235
System.out.println method, 143–144, 189

T

tab stops, 352–353
Table class, 417
Tatman, Rachael, 326
technical questions, websites for, 447
Technical Stuff icon, 4
telling variables where to go, 193–196
temperature
  appearance of, 214
  comparing, 226–228
  constructing, 215–217
  creating colder temperatures, 223–225
  default constructor, 217–219
  defining, 207, 208–211
  record class, creating, 220–223
  temperature scale, 207–208
  what you can do with, 211–214
Temperature class
  accessor methods, 208–211
  constructors, 216
  creating colder temperatures, 223–225
  default constructor, 217–218
  record classes, 220–223
  subclasses for, 270–272, 273–274
TemperatureCelsius class, 277–278
TemperatureForPercy class, 270–274
TempScale type, 207–208, 224–225
terminal operation, 340
test blocks, 111–112
test case, 177
test scores, tracking, 361–364
testing installed software, 22
text fields, adding to apps, 436–442
TheServerSide website, 447
this keyword, 167, 193, 216–217
three triangles, 145
throw clause, 380
throws clause, 378–381, 396, 399, 401
throws IOException clause, 244
TIOBE Programming Community Index, 451
Tip icon, 3
Tolstoy, Leo, 35
toString method, 222–223
toString method, inherited, 214
totalOfAverages field, 303

traditional comments, 51–52
Triangle class, 423–424
troubleshooting
  ArithmeticException, 392, 393
  catching exceptions, 380, 382
  catching two or more exceptions at once, 394
  checked exceptions, 396
  custom-made exceptions, 385–387
  errors versus exceptions, 394
  FileNotFoundException, 244–245, 395–399, 400–401
  general discussion, 378–381
  InputMismatchException, 241, 288
  matching catch clause, 387–393
  NumberFormatException, 379–381, 384
  NumberTooLargeException, 389–394
  OutOfRangeException, 386–387, 388, 389–394
  parameters in catch clauses, 382–385
  runtime exceptions, 386–387, 400
  throwing exceptions, 378–381
  throws clauses, 399, 401
  try–catch statements, 396–398, 401
  try-with-resources feature, 402–405
  unchecked exceptions, 396
try clause, 380, 382, 387–393, 396
Try it Out icon, 3
try–catch statements, 382, 384–385, 396–398, 401
try-with-resources statement, 402–405
Two Plus Two program
  main method, 43–44
  methods, 40–43
  outline forms, 46–50
  overview, 39–41
  statements, 44–46
twoPlusTwo program, 145–147
types
  boolean, 78–79
  byte, 69, 75
  char, 75–77
  with digits beyond decimal point, 65–66
  double, 64–66, 75, 100–101, 322
  enum, 207–208, 224–225
  float, 64–66, 75
  general discussion, 63–64
  int, 69, 75, 81, 99–100, 321–323
  long, 69, 75
  matching, 259–260
  overview, 61
  primitive
    ArrayList class and, 321–323
    arrays and, 357
    boolean type, 78–79
    char type, 75–77
    overview, 74–75, 407
    perimeter passing, 198
    wrapper classes, 322
  reference
    abstract classes, 417–424
    abstract methods, 424–425
    ArrayList class and, 321
    calling methods with, 94–95
    creating and referring to instances with, 93
    experiments with, 95–96
    general discussion, 407–408
    interfaces, 408–415
    overview, 90–91
    Random class, 91
    reserving variable names with, 91
    String class, 91
  return, 171
  short, 69, 75
  TempScale, 207–208, 224–225
  for whole numbers, 69

U

UN Security Council example, 30–31
unchecked exceptions, 396
underscores (_), 61, 66
unnamed packages, 283
update, for statements, 141–142
UseAccount class, 187–189
user groups, 446, 452–453
Utensil constructor, 275–276

V

vacancies, checking for, 25–26, 357–359
values
  defined, 61
  displaying, 67

filling arrays with, 351–352
inserting into arrays, 361–362
passing parameter lists by, 197–198
passing to methods, 174–175
returning from methods, 175–177
var declarations, 70, 149–150, 163, 350
variable declarations, 64, 69
variable name, 61
variables
amountInAccount, 60, 61
anEmployee, 238
assignment statements, 62
atomicCity, 193–196
balance, 158–159
call to print, 66–67
char, 148–149
combining declarations and initializing, 69–71
declaring, 160–161
evidence, 151
experiments with, 196–197
fields versus method-local, 191–193
functional programming and, 341–343
general discussion, 60–61
instance, 191
JShell, 71–74
largestSoFar, 355–356
mars, 191–193, 194–196
method-local, 191–193, 294–295
myAccount, 158–159, 161, 162, 167
name, 158–159
numbers without decimal points, 67–69
operators
assignment, 88–90
asterisk, 81
declaring variables, 83
increment and decrement, 83–88
minus sign, 80
plus sign, 79–80
remainder operator, 81–82
slash, 81
overview, 190
primitive types
ArrayList class and, 321–323
arrays and, 357
boolean type, 78–79
char type, 75–77
overview, 74–75, 407
perimeter passing, 198
wrapper classes, 322
reference types
abstract classes, 417–424
abstract methods, 424–425
ArrayList class and, 321
calling methods with, 94–95
creating and referring to instances with, 93
experiments with, 95–96
general discussion, 407–408
interfaces, 408–415
overview, 90–91
Random class, 91
reserving variable names with, 91
String class, 91
reply, 151–152
runningTotal, 356
smackoverAR, 200–202
smackoverARpop, 198–199
telling where to go, 193–196
types of values in, 63–64
types with digits beyond decimal point, 65–66
years, 69
yourAccount, 158–159
visitIdaho method, 194–196
visitNewJersey method, 195–196
visitPennsylvania method, 191–193
Visual Studio Code (VS Code), 22
void, 201

W

Warning icon, 3
web pages, generating from javadoc comments, 52–53
websites
for book, 445
Coderanch, 447
Garden State Java User Group, 446
InfoQ, 447
Java, 446
Java API documentation, 446
Java Community Process, 447
Java language specification, 446

websites *(continued)*
New York JavaSIG, 446
news about Java, 446
Oracle, 446
Stack Overflow, 447
for technical questions, 447
TheServerSide, 447
TIOBE Programming Community Index, 451
user groups, 446
`whatever` field, 216–217
`while` statements
`break` statements and, 152–153
`continue` statements and, 153
general discussion, 136–139
overview, 145–146
white space, 236
whole numbers, 69
`WordFrequencies.txt` file, 326–327
`Work` class, 299
wrapper classes, 321–323
writing arrays to files, 359–360
writing code, benefits of, 9–10
writing payroll checks, 237–242

X

XML (eXtensible Markup Language) tags, 435–436

Y

Y2K problem, 378
`years` variable, 69
yield app, 133
`yourAccount` variable, 158–159

About the Author

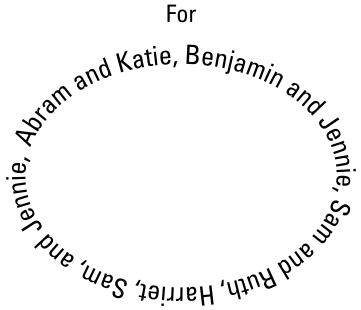
Barry Burd received a master's degree in computer science at Rutgers University and a PhD in mathematics at the University of Illinois. As a teaching assistant in Champaign–Urbana, Illinois, he was elected five times to the university-wide List of Teachers Ranked as Excellent by Their Students.

Since 1980, Dr. Burd has been a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. He has spoken at conferences in the United States, Europe, Australia, and Asia, and in 2020 he was honored to be named a Java Champion. He is the author of many articles and books, including *Beginning Programming with Java For Dummies* and *Flutter For Dummies*, both from Wiley.

Dr. Burd lives in Madison, New Jersey, with his wife of n years, where n > 44. In his spare time, he enjoys being a workaholic.

Dedication

For



Author's Acknowledgments

I was very lucky when the people at Wiley picked Paul Levesque to edit one of my books. His contributions to several of my works have been extensive and immeasurable.

Thanks also to Steve Hayes and Hanna Sytsma for their hard work and support in so many ways.

Thanks to Rod Stephens and Becky Whitney for their efforts in editing this book.

Thanks to the staff at Wiley Publishing for helping to bring this book to bookshelves.

Thanks to Frank Greco and the leaders of the New York JavaSIG. Thanks to Michael Redlich and the leaders of the Garden State Java User Group. Thanks to my colleagues, the faculty members in the Department of Mathematics and Computer Science at Drew University. As always, a special thanks to Richard Bonacci, Cameron McKenzie, Scott Stoll, and Gaisi Takeuti for their long-term help and support.

Publisher's Acknowledgments

Acquisitions Editor: Steve Hayes
Senior Project Editor: Paul Levesque
Copy Editor: Becky Whitney
Tech Editor: Rod Stephens
Production Editor: Tamilmani Varadharaj
Cover Image: © Henrik5000/Getty Images

Take dummies with you everywhere you go!

Whether you are excited about e-books, want more from the web, must have your mobile apps, or are swept up in social media, dummies makes everything easier.

Find us online!

dummies.com

dummies
A Wiley Brand

Leverage the power

Dummies is the global leader in the reference category and one of the most trusted and highly regarded brands in the world. No longer just focused on books, customers now have access to the dummies content they need in the format they want. Together we'll craft a solution that engages your customers, stands out from the competition, and helps you meet your goals.

Advertising & Sponsorships

Connect with an engaged audience on a powerful multimedia site, and position your message alongside expert how-to content. Dummies.com is a one-stop shop for free, online information and know-how curated by a team of experts.

- Targeted ads
- Video
- Email Marketing
- Microsites
- Sweepstakes sponsorship

20 MILLION PAGE VIEWS **EVERY SINGLE MONTH**

15 MILLION **UNIQUE** **VISITORS PER MONTH**

43% OF ALL VISITORS ACCESS THE SITE **VIA THEIR MOBILE DEVICES**

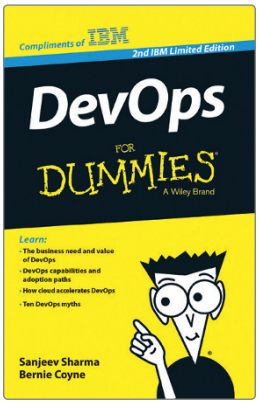
700,000 NEWSLETTER **SUBSCRIPTIONS** **TO THE INBOXES OF** *300,000* UNIQUE **INDIVIDUALS EVERY WEEK**

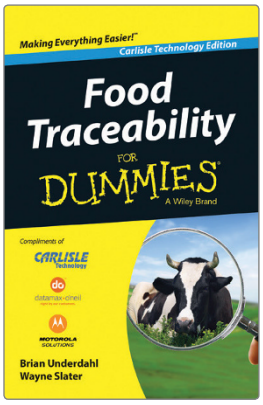
# of dummies

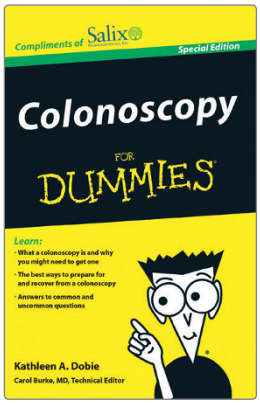
Custom Publishing

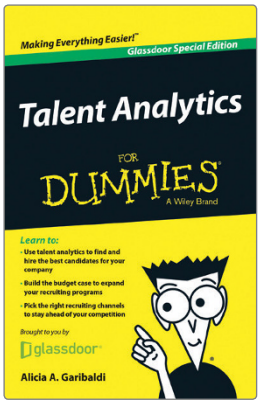
Reach a global audience in any language by creating a solution that will differentiate you from competitors, amplify your message, and encourage customers to make a buying decision.

- Apps
- Books
- eBooks
- Video
- Audio
- Webinars









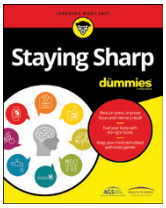
Brand Licensing & Content

Leverage the strength of the world's most popular reference brand to reach new audiences and channels of distribution.

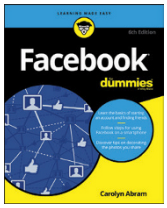
For more information, visit dummies.com/biz

dummies
A Wiley Brand

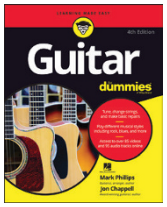
PERSONAL ENRICHMENT



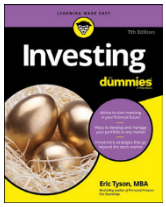
9781119187790
USA $26.00
CAN $31.99
UK £19.99



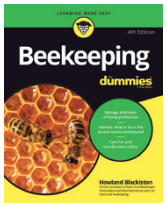
9781119179030
USA $21.99
CAN $25.99
UK £16.99



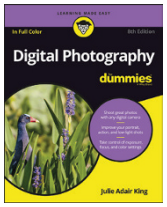
9781119293354
USA $24.99
CAN $29.99
UK £17.99



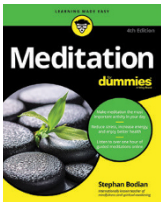
9781119293347
USA $22.99
CAN $27.99
UK £16.99



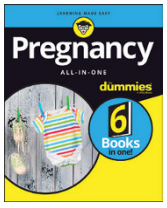
9781119310068
USA $22.99
CAN $27.99
UK £16.99



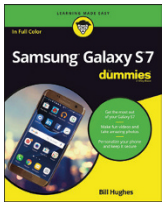
9781119235606
USA $24.99
CAN $29.99
UK £17.99



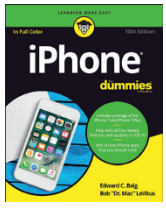
9781119251163
USA $24.99
CAN $29.99
UK £17.99



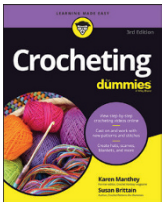
9781119235491
USA $26.99
CAN $31.99
UK £19.99



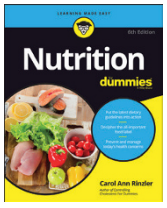
9781119279952
USA $24.99
CAN $29.99
UK £17.99



9781119283133
USA $24.99
CAN $29.99
UK £17.99

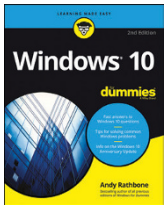


9781119287117
USA $24.99
CAN $29.99
UK £16.99

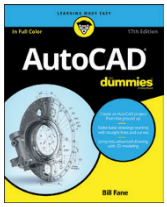


9781119130246
USA $22.99
CAN $27.99
UK £16.99

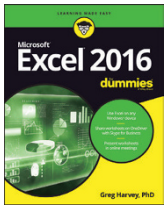
PROFESSIONAL DEVELOPMENT



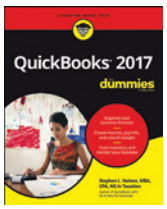
9781119311041
USA $24.99
CAN $29.99
UK £17.99



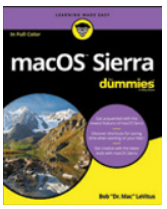
9781119255796
USA $39.99
CAN $47.99
UK £27.99



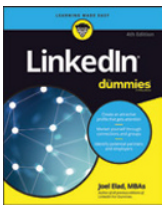
9781119293439
USA $26.99
CAN $31.99
UK £19.99



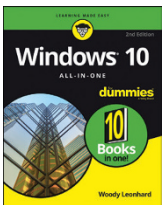
9781119281467
USA $26.99
CAN $31.99
UK £19.99



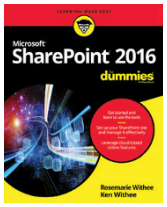
9781119280651
USA $29.99
CAN $35.99
UK £21.99



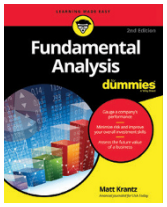
9781119251132
USA $24.99
CAN $29.99
UK £17.99



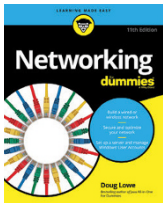
9781119310563
USA $34.00
CAN $41.99
UK £24.99



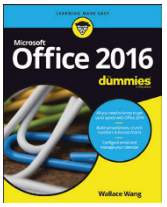
9781119181705
USA $29.99
CAN $35.99
UK £21.99



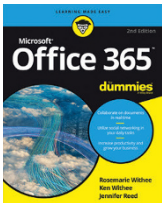
9781119263593
USA $26.99
CAN $31.99
UK £19.99



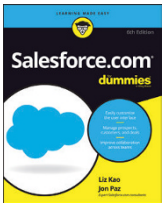
9781119257769
USA $29.99
CAN $35.99
UK £21.99



9781119293477
USA $26.99
CAN $31.99
UK £19.99



9781119265313
USA $24.99
CAN $29.99
UK £17.99



9781119239314
USA $29.99
CAN $35.99
UK £21.99

9781119293323
USA $29.99
CAN $35.99
UK £21.99

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.